FFDev: Progress Towards the Generation of *ab initio* Force Fields

by

Joshua Paul Radke
B.A., University of Minnesota, 1994

A thesis submitted to the Faculty of the
Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Chemistry
2002

This thesis entitled:
FFDev: Progress Towards the Generation of *ab initio* Force Fields
written by Joshua Paul Radke
has been approved for the Department of Chemistry

____
David M. Walba

____
Matthew A. Glaser

May 31st, 2002

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline

Radke, Joshua Paul (Ph.D. Chemistry)
FFDev: Progress Towards the Generation of *ab initio* Force Fields
Thesis directed by Professor David M. Walba

Classical interaction potentials, or force fields, are the fundamental input for any molecular simulation. Currently available force fields suffer from several limitations; namely availability, appropriateness, and quality. Low quality interaction potentials necessarily give low quality results when used in molecular simulations. The current state of force field development lies in the hands of a few specialists. Users of existing force fields are required to either purchase software, or implement their own software to use them. Also, if a user wants an improved force field, they are required to either start their own research program for that purpose, or wait for an update to an existing force field to be published. Furthermore, the procedures used to derive parameters for existing force fields (whether they are semi-empirical, or better yet, based on *ab initio* data) are often poorly documented. We have developed a suite of software that serves as a foundation for the on demand creation of strictly appropriate custom force fields from *ab initio* data. As the parameterization is automated, the element of human error/subjectivism in the many required transcription and decisions steps is eliminated. Further, every non-topologically/stereochemically equivalent atom has its own atom type and parameters. By employing software to do the force field creation from scratch, we have also created the opportunity for routine improvement of force fields by modifying the method of extraction of *ab initio* data, decreasing (and hopefully, eventually eliminating) reliance on experimental data. We believe that the best parameters for any classical interaction potential must come from *ab initio* data, and that our approach will eventually allow researchers to have access to free, fundamentally sound, appropriate, and highly accurate force fields.

Acknowledgements

This work was made possible by the generous support provided by NSF MRSEC Grant DMR 98-0555. More importantly, however, are the people who contributed directly to the completion of this thesis. Dr. David Walba, for his patience, and giving me the time to find a project that really tickled my fancy; Dr. Matthew Glaser, for the endless hours of brainstorming, direction, and instruction; but most of all for his patience with my (occasionally headstrong) demeanor. I would also like to thank Dr. Edgardo Garciá for his scientific and personal contributions.

For any (seemingly endless) journey, there are literally hundreds of people encountered along the way, and I would also like to thank them, especially any I miss in the rest of these acknowledgements.

Family is the most imporant support element one can have, and I would like to thank some of my family. First and foremost, I would like to thank my God for listening to my prayers over the years, which must have sounded like babbling to him. In close second comes my wife Stefani, who has been insanely patient with my continuously babbling about ideas no 'normal' people would care to hear about. I would also like to thank my father, Skip, for teaching me the curiosity to pursue a PhD, and my mother, Sherry, for who I am today. I'd also like to thank my brother, Adam, for all of the great Minnesota Vikings games Sundays. Thank you also to Leigh and Dana. And to the coolest in-laws on the planet, Calvin, Linda, Scott and Vickie, Rita, Bill, Patrick (Boogaman), Ricky, Scotty, and Cody.

Professionally, this project would not have been possible without the visionary movement started by Richard Stallman; open source software. Since the

inception of the Free Software Foundation, there have been numerous pioneers, who have made open, useful, software available to all, not just the wealthy. In particular, Linus Torvalds, Eric Raymond, Larry Wall, Bob Young, Kirk McKusick, and Tim O'Reilly.

My family back in Minnesota has been there for me every year (except the present one) for opening of fishing season, a grand time of fellowship. I'd like to thank Valerie, Wayne, Tim (yes, I'm done), David, Joe, and John; Butch, Ron, Tina, Amanda, and their families; and of course, Rossie. They're all Herschbachs, except for the Radkes that come from afar, Bud, Donna, Kurt, Mark, and Julie. Thank you also Rusty and family.

I have a couple of things I do to help preserve my sanity. The most important of these has been teaching. I would like to thank the Minority Arts and Sciences program at the University of Colorado. Thank you also Alphonse, Angela, and Wendy. The real people who deserve my thanks are the students, who have given me more than I could give them in a lifetime. I can't name you all, but thank you.

All of my co-workers during my time here have been helpful in some capacity. I'd like to thank Uwe, Forrest, Dan, Craig and Loretta (the wovewy), Bruce and Valerie, Eva and Bill, Ken and Tracy, Jen and Brian, Lei and Phong, Lixing, Matt and Darcie, Ethan and Jen, Alan, and Tim (for most of the illustrations in this thesis, as well as for his trivia savvy).

And finally, I'd like to thank all of my oldest friends, for believing I could do this, Patricia and Phillip (and family), Joe and James Waldo, Jim and Sue Hood, John and Brenda Fowler, Jon Ryan, Sean Flynn, Jim Miller, Brian Jarvis, Snuffleupagus

(do I win, Will?), Brent, Steve, Peter Tielemann (for the fun time, and the work talk), Murray and Alison, Dennis, Iotis, Sarah, Scott, Erik, Katrina, Jeannie, Jennicam Jen (no socks?), Jen, Jen (thanks for the car!), Otis and Felix, Marsha, Mark, Summer, Siri, Jeff, Laura, Chris, and the rest of the Fargo crew. And of course, who can forget the value and support Aluminum siding provides for all of us.

Thank you, all of you who have been a part of my life, mentioned or unmentioned you have my deepest gratitude for sharing my life with me.

Table of Contents

CHAPTER 1 1

Introduction 1

The Simple Background 1

The Simple Motivation 4

The real work 5

The real background and motivation 7

CHAPTER 2 17

The results 17

Compound 1 23

CHAPTER 3 26

Software, Algorithms, and Gory Details 26

Overview 26

Design 27

Conventions 32

On the topic of descriptors 32

The ubiquitous qcode 33

Functional overview 38

The generation system 43

Other libraries and utilities 46

CHAPTER 4 53

A tutorial 53

Getting Started 54

The path to patience 56

Completion 59

Closure 59

CHAPTER 5 .... **62**

What's New, revisited 62

High quality force fields of arbitrary forms from first principles 62

Background 62

Motivation 63

Procedures and Justification 65

Conclusions 68

CHAPTER 6 .... **70**

Wrapping it all up 70

Supplementary materials 70

Accomplishments 71

Future work 72

In closing … 73

BIBLIOGRAPHY **75**

APPENDIX A 80

Compound 1 80

APPENDIX B 91

Atom Map List for Compound 1 102

Bond Map List for Compound 1 104

APPENDIX C 106

cmap 107

finstr 108

makestr.pl 110

general 133

general/os_specific 151

genff 152

graveyard 156

log2str 158

one_timers 159

perl_modules 161

qdb 171

Force field creation programs 199

Miscellaneous Programs 213

qdb/qdb_maintenance_utilities 216

runff 227

shlib/CFUNCS 235

sim 237

Table of Figures

Figure 1: The structure of the Smectic C phase 2

Figure 2: The "back of the envelope" Boulder model 3

Figure 3: A molecule in two orientations in the Boulder model binding site 4

Figure 4: Illustrations of the tilt plane and symmetry of the phase............................ 6

Figure 5: Moore's Law, as shown for the Intel x86 series of processors. 9

Figure 6: The real accomplisment.. 14

Figure 7: The compound which force field was generated...................................... 18

Figure 8: Auto correlation for the simulation of compound 1 23

Figure 9: Polarization history for the simulation run on Compound 1 25

Figure 10: The "sphere of influence" 37

Figure 11: Collaboration summary diagram for the software.................................. 39

Figure 12: Exact partial match vs. 'Chemists Intuition' 45

Figure 13: The program fffront.pl.. 50

Figure 14: A rendering of Compound 1 from molren.pl.. 52

Chapter 1

Introduction

Every great endeavor begins with a story. In this case, it turns out that the eventual goal was much different than what was really done. At the inception of this project, I was asked to do a 'single molecule in a binding site' calculation. Upon studying the problem, it quickly became apparent that actually completing this goal would be a long process, and the expertise gained would be only applicable to the person who actually did all of the work. I wanted a 'permanent' solution to this problem, and so began FFDev.

The Simple Background

Liquid crystals are molecules that organize themselves in such a way that they have properties of both liquids, and crystals. They are truly liquids in that they flow, and take the shape of their container, but they also display some degree of long range positional or orientational order (but never enough to be identified as crystalline solids). This work focuses only on molecules within the smectic C phase (**Figure 1**).

The Boulder model [1] for polarization is really quite simple, and doesn't necessarily even require a computer to apply. Empirically, it was discovered very early [2] that for molecules within the smectic C phase, the flexible tails are always more tilted than the rigid cores. This simple fact implies that molecules within a smectic C phase prefer to be (or are at their lowest free energy) in conformations and orientations that fit well within the Boulder model binding site (**Figure 3**). The shape

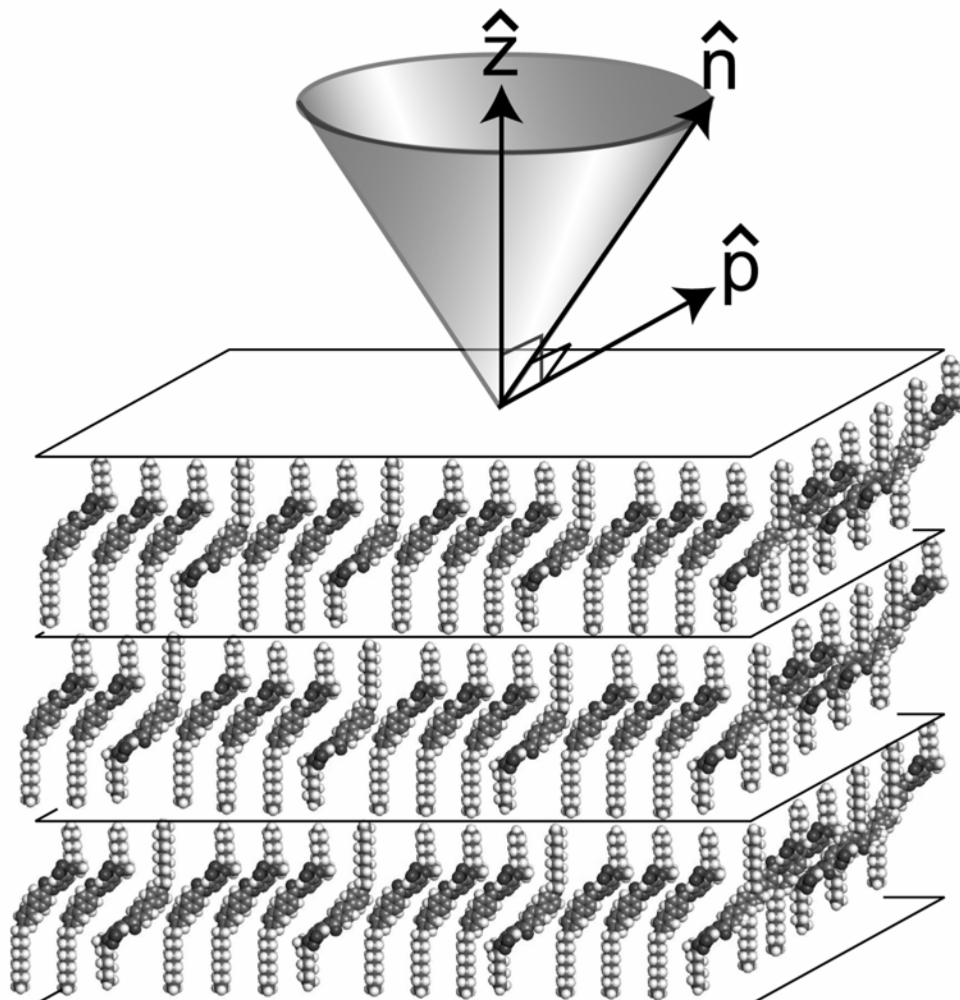


Figure 1: The structure of the Smectic C phase. The cone at the top of the figure demonstrates the important directors of the Smectic C phase. The vectors **n** and **z** define the tilt plane, which is perpendicular to the polar axis **p** (in the case of a Smectic C* phase).

of the Boulder model binding site represents the effect of the Smectic C phase on a single molecule. One can easily apply this model by doing a bit of simple drawing, as shown in **Figure 2**

If one hopes to get quantitative, as opposed to qualitative, information from this model, they must convert this simple idea to an algorithmic basis. Maier-Saupe

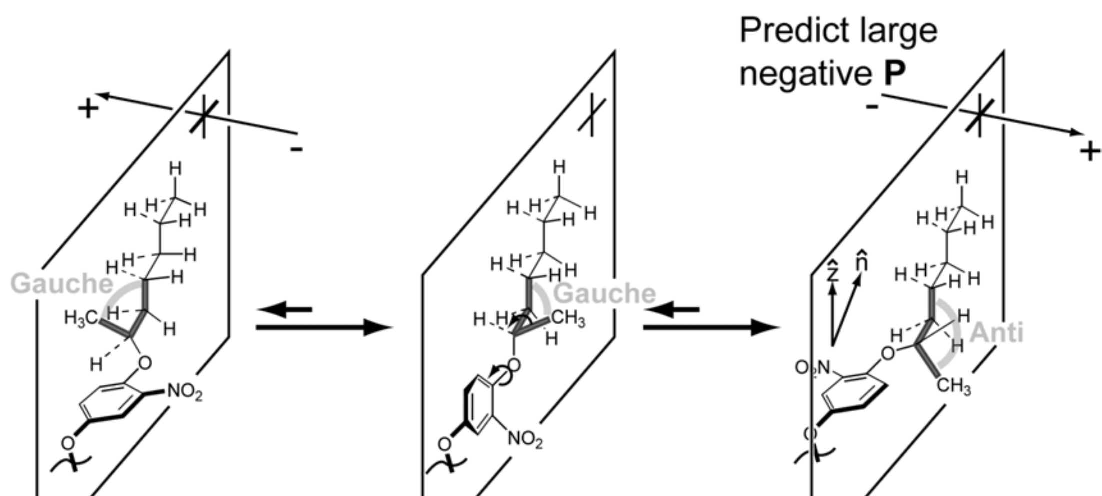


Figure 2: The "back of the envelope" Boulder model. Begin by placing the tails more tilted than the core (in an all trans configuration), and putting the molecule in a conformation that is intuitively low energy (center pane); then simply "crankshaft" around the indicated torsions to generate other geometries.

mean field theory [3] provides the perfect model for doing so. By docking the molecules into the Boulder model binding site, we are saying that there is some energy cost for deviating from conformations that fit well within that binding site. The true source of these energy costs need not be established, as we say they result from the sum of inter-molecular forces within the Smectic C phase.

The distribution function of a molecule is defined as the 'population', or probability, of every possible molecular configuration [4]. While the true distribution function of a molecule (at a given temperature and pressure) can never be found exactly, except for the simplest of cases, some methods exist to allow us to get very good approximations of this distribution function.

It is important to emphasize here that we are dealing with real liquids. A molecule in the Smectic C phase still has a great deal of conformational flexibility,

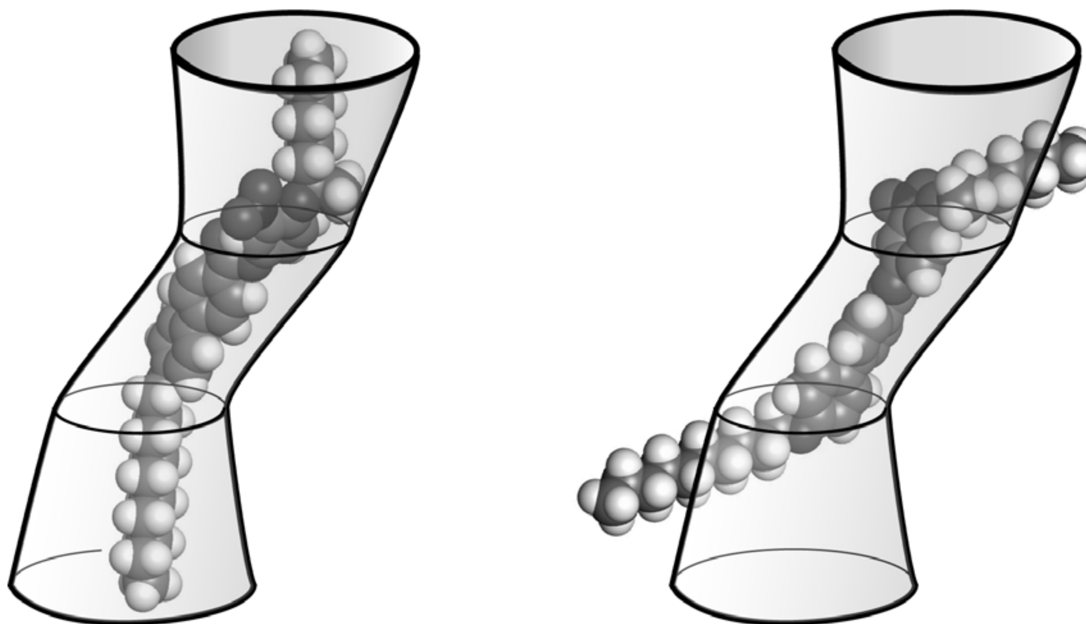


Figure 3: A typical molecule in two orientations in the Boulder model binding site. Two orientations are shown; in the left one, there would be no additional energy cost, in the right one, (with the same molecule simply rotated about its core axis), the molecule would suffer a significant energy penalty). The shape of the binding site simply comes from the influence of the rest of the phase upon a single molecule.

and should never be thought of in the way that we typically think about crystalline solids.

In order to calculate this distribution function, we need to be able to evaluate the energy of all of the possible molecular configurations. Someday in the very distant future, we will be able to get 'arbitrarily exact' energies for all possible configurations, via *ab initio* calculations. Until that day comes, we instead must rely on a classical energy expression to evaluate these values. Our quantitative version of the Boulder model adds energy costs for deviations from the binding site geometry, as shown in **Figure 3** (specific details are presented in Chapter 2).

The Simple Motivation

The origins of the macroscopic polarization observed within smectic C* (the * simply means the phase is composed of chiral molecules) phases are one of the properties that we most understand. In 1974, Bob Meyer [5] gave a most elegant symmerty argument for why this macroscopic polarization should exist, but no work to date can accurately predict, *a priori*, what the experimental value of the macroscopic polarization should be for a given molecule. It's clear that the answer must exist, and that it involves the distribution function of a macroscopic sample, but once again that problem space is far too large to handle with both accuracy and precision (if we are challenged to find the distribution function of a single molecule, finding the distribution function for a macroscopic sample of liquid crystal molecules is completely out of our reach!)

Using the Boulder model to calculate the distribution function for a single molecule in this binding site allows us access to several numbers, including the overall (average) dipole moment of the molecule. The true macroscopic polarization is actually a polarization density (i.e. nanocoulombs/cm^2, or debye/cm^3), so we can take the dipole, and divide by the volume of the molecule, as derived from the bulk density of the liquid crystal. There is one further refinement necessary before we can report a polarization density, and that is to take the calculated dipole, and find the component of it along the 'true' polar axis, which is defined by the symmetry of the phase (**Figure 4**); we use this vector instead of the original dipole.

The real work

It turns out that the most difficult part of the previously outlined procedure is to get 'useful' force fields (interaction potentials) that correctly reproduce the shapes and energies of the various conformations in the distribution function. Many force

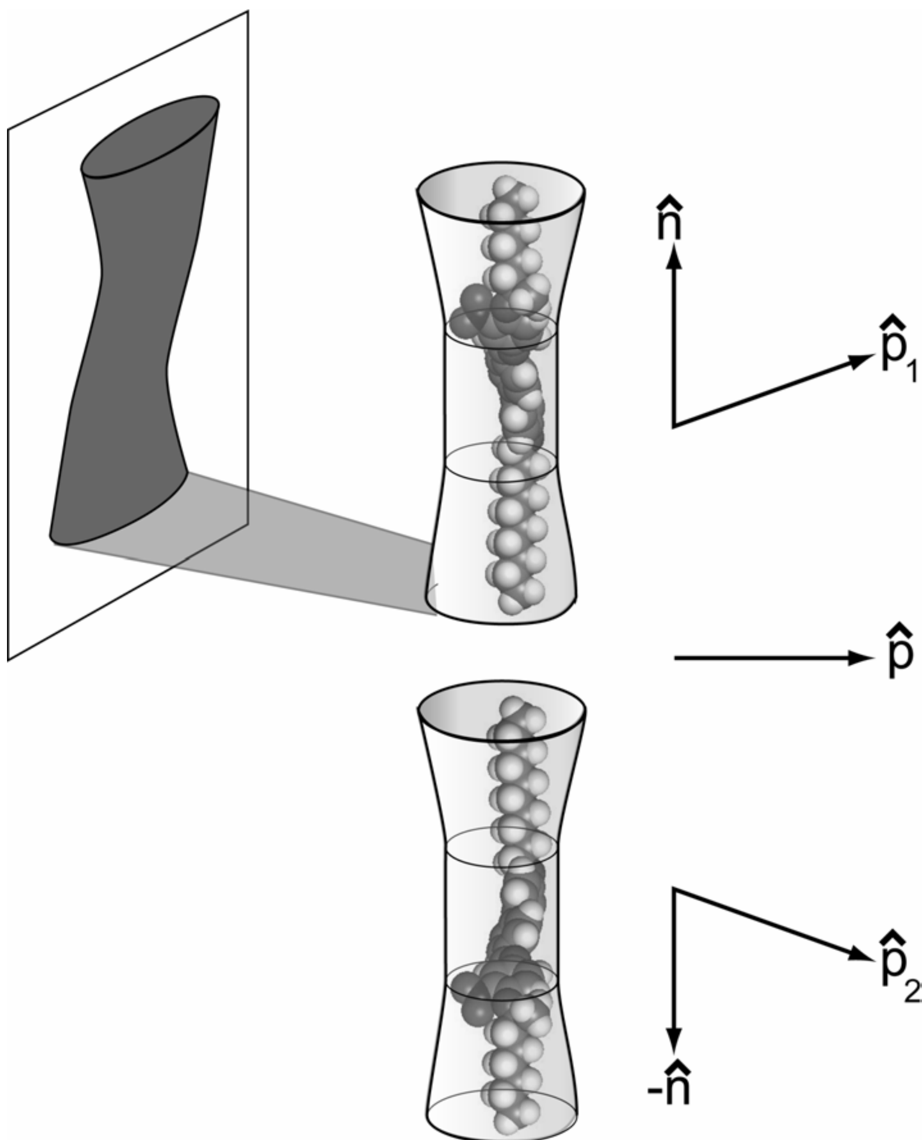


Figure 4: Illustrations of the tilt plane, which is perpendicular to the polar axis (top left), and the symmetry of the phase (right). In all cases of known calamitic liquid crystals, **n** goes to -**n**, meaning that there is no polar order along the long axis of liquid crystal molecules. In the case of molecules in the Smectic C phase, this rule manifests itself by enforcing a C_2 symmetry axis parallel to the polar axis. As a result, any component of polarization within the tilt plane in a binding site calculation will go to zero when the -**n** conformations are taken into account.

fields already exist [6], but none of them diligently reproduce energies associated with the many dihedrals in any given liquid crystal. Not surprisingly, the Boulder model is very sensitive to molecular shapes, which are in turn very sensitive to these torsions.

For the aforementioned reasons, we are required to create our own force fields from *ab initio* data. Previous work at the University of Colorado had done exactly that [7], but again, the process was arduous and error-prone. We wanted to develop a software system to automate the majority of the process for many reasons. Firstly, it would greatly decrease the (human) time involved with creating one of our custom force fields. Secondly, it would remove the possibility of errors associated with transcription of values. Finally, it would algorithmically define procedures, and remove (or at the very least regularize) the many human decisions involved with the process. FFDev is the culmination of that effort, and the focus of the rest of this thesis.

The real background and motivation

The FFDev project endeavors to support and grow a relatively recent marriage in the world of physical chemistry. The fields of quantum chemistry and statistical mechanics have already begun to merge in the field of computational molecular mechanics, a marriage that promises profound affects in the very near future. Due to factors discussed shortly, the rate of progress in this field has been, and promises to continue to be, phenomenal.

Quantum chemistry

Quantum chemistry was first introduced by Heisenberg in 1925 [8, 9]. In the very same year, it was given a matrix-algebra formulation by Born and Jordan [10]. In 1926, Schrödinger independently introduced his wave mechanics formulation, proved the equivalence of the two methods, and established his name in history [11] (primarily due to the simpler mathematical formulation). While a firm footing to real solutions of atomic systems had been established, it became quickly apparent that systems with any real complexity were unsolvable. Dirac's famous quote set the stage for the future of quantum chemistry to date, he said [12]:

> "The underlying physical laws necessary for the mathematical theory of a large part of physics and the whole of chemistry are thus completely known, and the difficulty is only that the exact application of these laws leads to equations much too complicated to be soluble."

Dirac's realization marks the beginning of computational chemistry (many mark the beginning by Pople's early work [13], but the fundamental problem in my mind is reducing the complexity of problems to manageable sizes). In order to solve any problem of greater difficulty than a single hydrogen atom as the sole member of a universe, approximations need to be made. Computational quantum chemistry is the field of making those approximations, and applying the resulting methods to solve real world problems. Solutions to electronic and nuclear structure of molecules are called *ab initio* results, meaning they're derived from first principles, and rely on nothing more than the specification of the system in question, and certain universal constants. Two factors have been responsible for recent rapid advances in this field. Moore's law [14] states that "the number of transistors per square inch on integrated

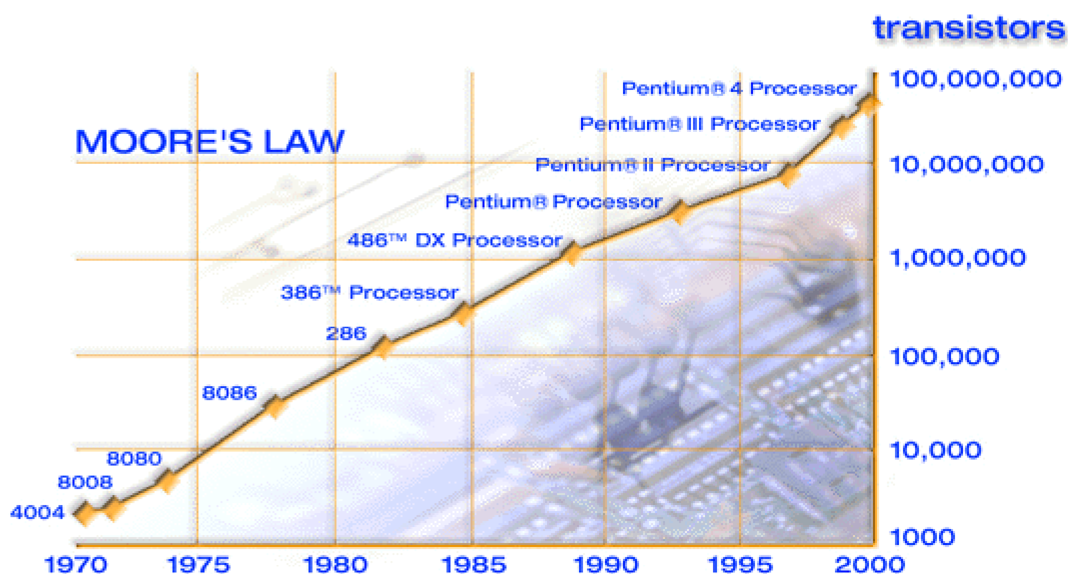


Figure 5: Moore's Law, as shown for the Intel x86 series of processors.

circuits doubles every 18 months". Transistor density quite closely translates to computational power/price (**Figure 5**, [15]).

I would have to say that the second major factor is the involvement of Industry and Academia in providing software to implement computational quantum chemical methods on modern computers. These two factors make is relatively easy to get high quality electronic and conformational structure information for moderate sized structures (on the order of 25 heavy atoms) in times ranging from several hours, to a maximum of a week, in most cases. The availability of solutions with such precision is one half of the fortunate timing that makes projects like FFDev possible.

Statistical mechanics

If quantum chemistry embodies the 'genius of the twentieth century', the field of statistical mechanics embodies a philosophical journey though the ages. Much of the following account is taken from "Sketching the History of Statistical Mechanics and Thermodynamics" [16]. Democritus (470 to 360 BC) is frequently credited to be the 'father of atomism'. Atomism is the concept that at some level, the universe must be composed of indestructible, discrete units. While his philosophy was soon 'trampled' by the horde of Aristotelians to come, his fundamentally correct postulation of the nature of matter would serve as the foundation for statistical mechanics. Around 150 BC, Hero of Alexandria wrote "Pneumatics", a fascinating (and definitely recommended reading!) book on the behavior of fluids, including air [17]. It appears that a 1575 translation of "Pneumatics" to Latin may have been at least partially responsible for the explosion of understanding to happen within the next 100 years, which would eventually lead to various formulations of the ideal gas law by Boyle, Charles, Gay-Lussac, and Avogadro by the early 1800's. In 1843, Waterston [18] published a complete kinetic theory of gases, but was ignored, though he later tried to publish his work in journals as well. It wasn't until 1884 that Gibbs coined the term 'statistical mechanics' to refer to the study of thermodynamic properties of systems by the application of kinetic theory.

By this time, matter was treated as atoms, and classical physics was used to derive the properties of large systems, by assuming certain things about the behavior of atoms within these systems. The predictive value of the kinetic theory of gases and the ideal gas law (along with its associated variants, such as the van der Waals

equation), are testament to the value of ‘simple’ classical models as powerful predictors of real phenomena. In the solution of all statistical mechanics problems, the single cohesive element is that the individual members of the system are given some behavior to govern their states, and the system is statistically analyzed, either in a time dependent, or time independent fashion. This analysis requires an integration or sum over all of the states. While some systems scale very well under this treatment (analytical expressions can be derived for any interesting property at arbitrary system sizes and/or timescales), the vast majority of conceivable problems do not, and so had been ignored until the advent of modern computers.

By the early 1950’s, there was significant effort being put forth [19] in the academic community to use ‘electronic computers’ to solve statistical mechanics problems that had been completely out reach of statistical mechanics until the emergence of computers.

The marriage

There’s no truly good way to draw the lines of when the marriage between quantum chemistry and statistical mechanics took place. As mentioned previously, computer simulations of systems of hard spheres were being done in the 50’s. The earliest ‘atomistic’ simulation may mark this beginning just before the end of the decade, and was published in 1960 [20]. By the 1970’s, computers were becoming powerful enough to treat systems of much greater complexity than simply collections of spheres. This marked the beginning of atomistic molecular mechanics as we know it today [21]. Despite the great variety of approaches to molecular modeling to date [6], a couple of key elements remain constant in all of the solutions.

Every force field must have some formula by which the energy of the system as a function of the positions of all of the members of the system can be evaluated. Ideally, that function will also have analytical derivatives of the energy with respect to the positions of each of the elements (this is necessary for expedience in time dependent simulations). A typical force field is simply a sum of terms, with each term providing the energy associated with a particular type of molecular feature. For every type of interaction in the entire system (i.e., for a bond comprised of two different types of atoms) we require parameters, or numbers that give information about that particular entity, such as bond length, and how 'strong' the bond is. The details of our implementation are beyond the scope of this thesis, though we'll provide an overview in Chapter 2; however, the input that goes into the force field is the prime focus.

For every force field known to the author, each atom in the system is assigned an 'atom type'. This is a descriptor whose purpose is to encapsulate all of the behavior of that atom in a variety of roles (i.e., atomic charge, as a member of a bond stretch or angle, etc.). This is a very useful concept, and allows us to treat systems with large numbers of atoms with a reduced number of parameters. Despite its general usefulness, this particular approximation seemed inadequate for our purposes. As a partial solution to this problem, we have implemented a combination of our own descriptors (discussed in detail in Chapter 3) and our own stereochemical descriptors for tetrahedral stereogenic carbons as a way to assign unique identities to all atoms that are not topologically and stereochemically equivalent.

The process of providing all of the parameters for a given simulation is unsurprisingly called parameterization. Of the existing force fields, there are two sources of parameters. One type derives the parameters empirically, i.e., they seek a certain outcome of a molecular mechanics calculation by changing the parameters until the desired answer is achieved. The other source of parameters is from *ab initio* calculations. The majority of existing force fields use some combination of the two approaches, generating semi-empirical force fields. Increased reliance on *ab initio* data for parameterization of force fields has led to some very high quality force fields [22], and has improved quality altogether, yet the problem still remains that the product force field is either too specific to be generally useful, or too general to provide correct parameters for every term in the force field (the number of atom types is significantly less than the number of distinct atoms in the molecule for which the force field was generated).

So what is it that distinguishes our force fields from all of the others that are available? To our knowledge, we have the only system designed to generate force fields <u>directly</u> from *ab initio* data, as well as provide a unique identity (atom type) to each and every non-topologically/stereochemically equivalent atom. Our current progress does not allow us to extract all of the desired parameters from *ab initio* data, but it does allow us to get the ones we are particularly concerned with (energies about dihedrals), and it further serves as a "proof of concept" that such a direct mapping can be accomplished, and may indeed be done by future work on this project.

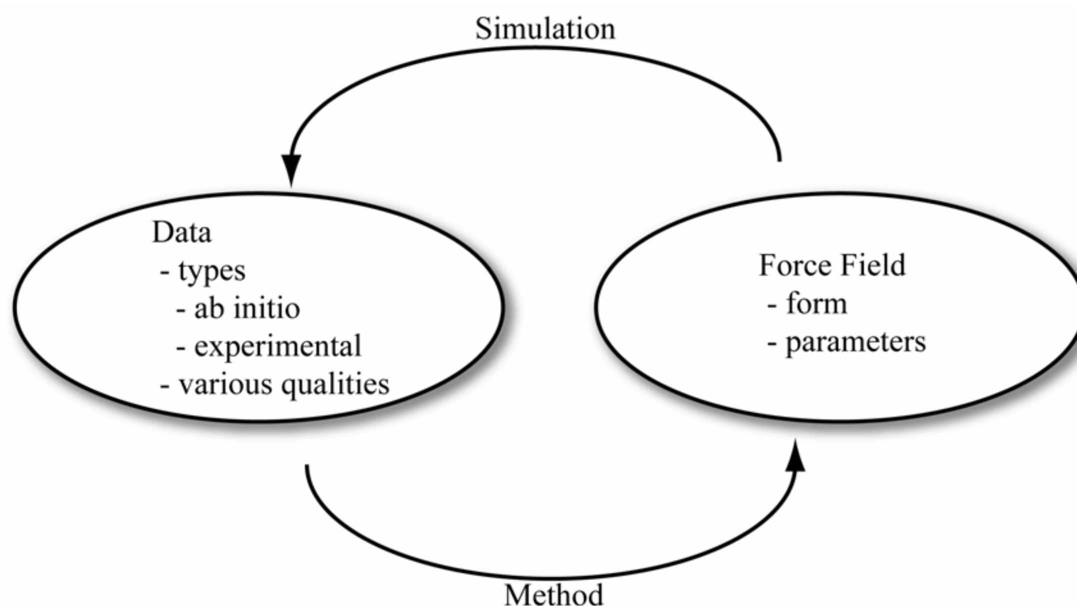


Figure 6: The real accomplisment. Creating and refining force fields is an iterative process. Current experts in the field have access to all of the tools to both run simulations with their force fields, and refine them using their own methods. Unfortunately, tools which automatically create force fields by well defined (and customizeable) methods have been largly unavailable until now.

It seems obvious that the best possible classical interaction potentials must have a direct relationship with electronic structure, as provided by ab initio calculations. What the specifics of this relationship are, however, is not clear at all. In the absence of our approach, the only way to get 'better' force fields is to either change the form of your existing one (most frequently by adding coupling terms, but almost always one ends up adding more parameters), or try harder to change the parameters that go into it, in an effort to get more accurate results. We have added a third approach to rapid systematic improvement, and that is to refine one's method of generating the final force fields. This approach many advantages. Primarily, it allows for rapid prototyping and refinement of force fields (**Figure 6**). More specifically, it affords us the opportunity to do our refinement not only by changing the method of data abstraction, but also by changing either paramters or the form of

the force field emprically, should we wish to. In short, we've opened the door for many more users to be involved with force field refinement. Finally, it promises to provide us with insight into the subtleties of why all classical interaction potentials fail, at some level.

A measure of success

No theory or model can be considered useful unless it is capable of reproducing (or better yet, predicting) experimental results. While the primary accomplishment is a "proof of concept", we still need to be able to verify its usefulness in a simulation. Since we are a liquid crystal group, and obviously supporters of the Boulder model, evaluation of our force fields within the context of that model seems the natural choice.

Van Gunsteren, et. al. [23] set out very clearly the elements of molecular mechanics simulations that must be considered, in order to validate the results. In that paper, he outlines five barriers to validation, and five basic requirements necessary to overcome those barriers. These are as follows:

1. A full description of the model and algorithms must be readily available.
2. A full description of the interaction function or force field must be readily available.
3. Simulation results must be shown as a function of simulation length.
4. The source code of the software must be able to be checked.
5. The set up of the simulations must be described in detail.

We have done our best to fulfill all of these criteria in presenting our results in Chapter 2. Classical interaction potentials were generated for a single test compound,

and the aforementioned ‘single molecule in a binding site’ calculations were done. The results of these test cases clearly demonstrate that we have generated reasonably good agreement with both previous simulations, and experiment.

Chapter 2

The results

Three 'single molecule in a binding site' calculations were finished and the results of those calculations are presented here. Before presenting the results, however, we'll discuss a bit more background on the details of how the calculations were done. The single test compound will be named Compound 1 throughout the rest of the thesis. The structure of these molecules can be seen in **Figure 7**.

The exact procedure used to generate the force field for the test compound is the subject of Chapter 4, which is more or less a 'walkthrough' chapter. Here we summarize the details of the input to the force field, and how these *ab initio* values were mapped onto our classical interaction potential.

Since the input molecules for our simulations are too large to handle with reasonable detail with *ab intio* calculations (**Figure 7**), and we wanted to develop a system to be useful for <u>any</u> sized input molecule, we did calculations on smaller fragments. Appendix B shows which fragments were used for each of the test cases. The procecure for generating these fragments is called fragmentation, and will be used in several other places in this text.

All *ab initio* calculations were performed with Gaussian 98 [24]. For each of the fragments, first the geometry was optimized using Becke's three parameter Hybrid Functional Using the LYP correlation functional with closed shell restricted

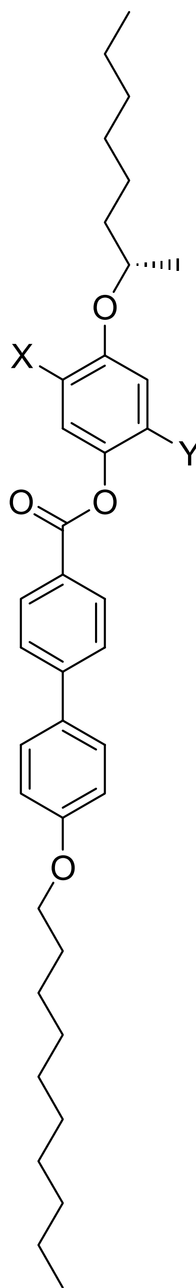


Figure 7: The compound for which force field was generated.

wave functions [25], with a 6-31g(d) basis set (RB3LYP/6-31g(d)). The energy of the optimized conformation was evaluated at the RB3LYP/6-311+g(2d,p) level. The terminology for the this procedure is to simply state that the energy was evaluated at the RB3LYP/6-311+g(2d,p)// RB3LYP/6-31g(d) level of theory.

For each of the dihedral vs. energy profiles we needed, at least 24 individual energies were calculated at the RB3LYP/6-31+g(d,p)//RB3LYP/6-31g(d) level of theory, which has been shown to give excellent results for its computational cost [26]. When optimizing the geometries of the fragments during the torsion scans, all dihedrals about certain bonds were frozen at their global energy minimum values. This is done to prevent dihedral vs. energy profiles which exhibit 'un-natural' asymmetry (based on the direction the profile is scanned), which frequently occur with unconstrained dihedrals. If a bond was between two sp^3 hybridized heavy (non-hydrogen) atoms, neither of the atoms was a terminal CH_3 group, and both of the atoms had no resonant (bond order 1.5) bonds, then all dihedrals about that bond were frozen.

For our purposes, we use fragments with hydrogens on sp^3 carbons absorbed into those carbons. The exact form of the force field that we use for our simulations is as follows [7]:

$U(\mathbf{r}^N) = U_{str} + U_{bend} + U_{tors} + U_{inv} + U_{vdw} + U_{coul}$, where the individual energy terms are defined as follows:

$$U_{str} = \sum_{\substack{bonds \\ ij}} \frac{1}{2} k_r \left(r_{ij} - r_{eq}\right)^2$$

$$U_{bend} = \sum_{\substack{angles \\ ijk}} \frac{1}{2} k_\theta \left(\theta_{ijk} - \theta_{eq}\right)^2$$

$$U_{tors} = \sum_{\substack{dihedrals \\ ijkl}} \sum_{n=0}^{6} c_{n\phi} \cos^n \phi_{ijkl}$$

$$U_{inv} = \sum_{\substack{umbrellas \\ ijkl}} k_\psi \left(\cos\psi_{eq} - \cos\psi_{ijkl}\right)^m$$

$$m = \begin{cases} 1, \psi_{eq} = 0 \\ 2, \psi_{eq} \neq 0 \end{cases}$$

$$U_{vdw} = \sum_{i<j} 4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right]$$

$$U_{coul} = \sum_{i>j} \frac{q_i q_j}{r_{ij}}$$

In the present work, all 1-2, 1-3, and 1-4 interactions are omitted in the evaluation of U_{vdw} and U_{coul}. The internal coordinates r_{ij}, θ_{ijk}, ϕ_{ijkl}, and ψ_{ijkl} are defined by:

$$r_{ij} = \left|\mathbf{r}_{ij}\right| = \left|\mathbf{r}_j - \mathbf{r}_i\right|$$

$$\theta_{ijk} = \cos^{-1}\left[-\frac{\mathbf{r}_{ij} \cdot \mathbf{r}_{jk}}{r_{ij} r_{jk}} \right]$$

$$\phi_{ijkl} = \cos^{-1}\left[\frac{\left(\mathbf{r}_{ij} \times \mathbf{r}_{jk}\right) \cdot \left(\mathbf{r}_{jk} \times \mathbf{r}_{kl}\right)}{\left|\left(\mathbf{r}_{ij} \times \mathbf{r}_{jk}\right)\right| \left|\left(\mathbf{r}_{jk} \times \mathbf{r}_{kl}\right)\right|} \right]$$

$$\psi_{ijkl} = \sin^{-1}\left[\frac{\mathbf{r}_{ij} \cdot \left(\mathbf{r}_{ik} \times \mathbf{r}_{il}\right)}{r_{ij} \left|\mathbf{r}_{ik} \times \mathbf{r}_{il}\right|} \right]$$

ψ_{ijkl} measures the angle between $\mathbf{r}_{ij}$ and the plane defined by $\mathbf{r}_{ik}$ and $\mathbf{r}_{il}$ for all three coordinate atoms i. The total inversion potential is taken to be the average of the umbrella torsion terms for the three possible choices of the special bond $\mathbf{r}_{ij}$. The definition of dihedral angle was as described by Kline and Prelog [27]. The total rotational potential about a bond is taken to be the average of all possible dihedrals about that bond. Both of these conventions were taken from the Dreiding II force field [28].

Parameters for bond stretching, and angle bending, were generic (Dreiding II [28]), though the equilibrium values for all bonds and angles were extracted from the global energy minimum conformation of the corresponding fragment. Generic inversion parameters were also used [28]. Point charges on the atoms were assigned based on the CHELPG scheme [29], and mapped onto the parent molecule from the relevant fragments. Carbons with absorbed hydrogens were assigned the sum of the charges of the carbon and all absorbed hydrogens. Van der Waals parameters were taken primarily from OPLS [30], though the values for the carbons containing absorbed hydrogens were taken from other sources [31].

This leaves only the parameters for the dihedrals to be determined. These parameters are determined in much the same way that they were in previous calculations of this type [7]. First, all torsional parameters are set such that no dihedral angle makes any energy contribution. Secondly, the torsion we are fitting is driven in exactly the same way as it was for the *ab initio* torsional potential (the same dihedral angles are driven, the same dihedrals are frozen), and the energy of the classical force field is recorded. Thirdly, for each dihedral angle, the difference

between the classical energy and the *ab initio* energy is recorded. Finally, the $c_{n\phi}$ parameters are fitted to reproduce this energy profile.

In a separate verification step, the newly found parameters for the torsion are used (instead of all being set to 0), and the classical energy is evaluated by driving the system in the same was that it was in the generation step. Appendix A contains graphs comparing the *ab initio* vs. classical energy for every fitted torsion, as generated in this verification step, along with an illustration of which dihedral was driven. Appendix B shows the fragmentation of the test compound, as well as a graphical representation of which atoms and bonds were mapped from the child to the parent compound.

As mentioned in the introduction, the Boulder model binding site calculations require that we algorithmically implement the empirical fact that the tail is more tilted than the core. To do this in a molecular mechanics calculation, we require only three parameters. Firstly, we define which regions of the molecule are tail, and which regions are core. Secondly, we define an angle between the core director and the tail director (which can be parametrized from experimental data, if desired). Finally, we need to add an elogation potential. Since the single molecule simulation is done in a vacuum, and we only penalize the tail for not being parallel to the tail director, omitting this elongation potential results in many conformations where the tail is folded, which is not in keeping with the Boulder Model.

To generate the molecular distribution function, we use a hybrid Monte Carlo scheme. Monte Carlo techniques generate the distribution function by evaluating the energy of trial states, and accepting or rejecting the state based on a Maxwell-

Boltzmann criteria. To generate the trial states, we provide the indiviual atoms with random velocities, and evolve the system with molecular dynamics. Molecular dynamics simply integrates the equations of motion, based on the energy terms in the final force field, and evolves the system. The number of molecular dynamics steps between trial configurations is chosen such that the auto correlation with respect to polarization (the property of interest to us) decays at an 'acceptable' rate.

Compound 1

Development of the force field for compound 1 (**Figure 7**) was quite routine, with the exeption of two torsions that are strongly coupled, and required care in parametrization. By inspection, it's easy to see that the torsion about the carbon-

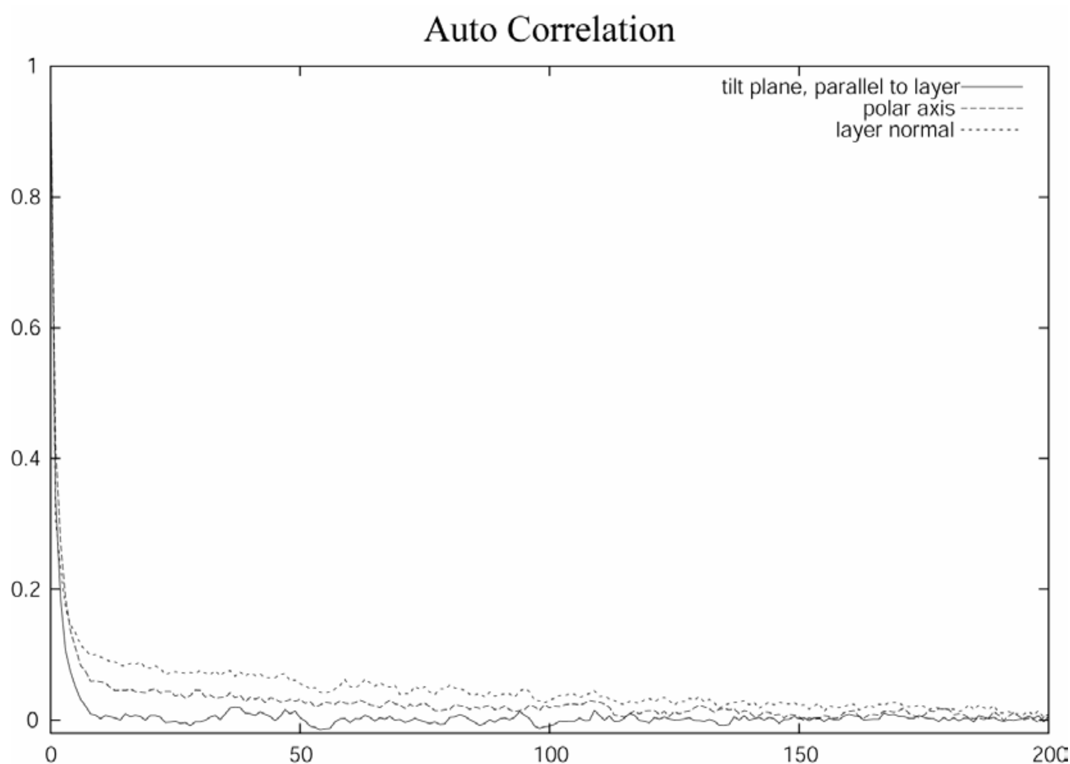
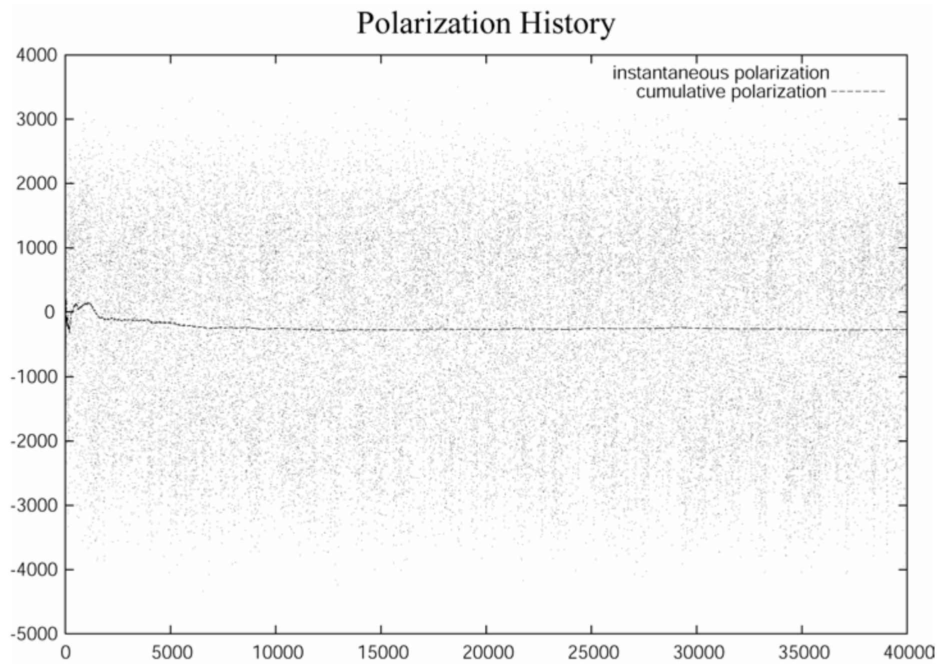


Figure 8: Auto correlation for the simulation of compound 1.

nitrogen bond in the nitro group, and the carbon-oxygen bond in the nearby anisole group definitely interact. Since it would be 'unphysical' to lock the nitro group into some conformation while getting *ab initio* energies for various anisole dihedrals, it was allowed to 'float'. This required us to first fit the nitro torsion, and then allow it to float while fitting the anisole torsion. The results of these fits can be seen in Appendix A.

The calculated polarization for Compound 1 was -260.277 ± 17.586 nC/cm^2, which is in good qualitative agreement with the experimental value of -550 nC/cm^2. While the simulation generates an absolute magnitude of the polarization which is less than the experimental value, this is to be expected, as the Boulder Model does not account for orientation of the cores along the polar axes (due to π stacking, steric, or some other intermolecular effects), which we anticipate would raise the magnitude of the polarization significantly.

Figure 8 shows the auto correlation function for polarizations along the three primary axes. This function tells us how long (or how many monte carlo steps) it takes for a given value to be decoupled from a previous one. **Figure 9** shows the polarization as a function of trial configurations.



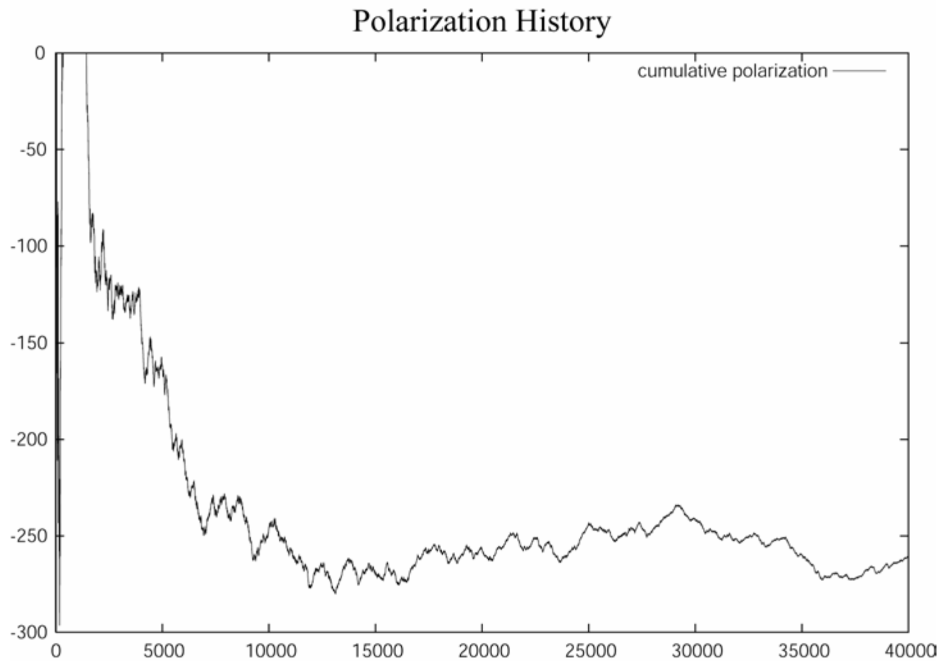


Figure 9: Polarization history for the simulation run on Compound 1, with instantaneous polarization (top pane), and without.

Chapter 3

Software, Algorithms, and Gory Details

This chapter covers the actual software that comprises FFDev. There are many programs in the entire suite, all with their individual functions. Here we do not try to give any kind of tutorial on using the software, as that is the subject of Chapter 4. Instead, we discuss the overall design, and the functionality of the individual components.

Overview

The rest of this chapter is definitely 'gory', so a brief summary of what happens when we generate a force field is presented here. The software is comprised of two agents (**Figure 11**). The qdb (quantum chemistry datbase) and related programs are resposible for holding, calculating, and returning various *ab initio* information to the process agent. The first step is to create a structure of the molecule for which you want a classical interaction potential. This can be done with any molecular drawing program, but for the best results, it should be in a conformation close to the global energy minimum. Running qdb_check on this file will create a partial force field file that contains all of the information presented in Appendix B (which atoms in the parent molecule should be represented by which fragment atoms, and the same information is available for all of the bonds). If the database did not have fragments for the initial parent molecule, another program will submit the *ab*

inito calculations. Finally, when the database has the necessary information for all of the fragments, the final force field is generated with the final programs.

Design

All too often in academic software development, the first step of software creation is neglected to some degree. That step is software design. There is no *de facto* authority on the subject, as it is still very much evolving [32], but browsing various sources does reveal a pattern of topics that are very useful to guide a non-software engineer in this step. Among the many considerations, we paid special attention to (and will discuss in more detail) the following:

1) Portability
2) Scalability
3) Usability
4) Maintainability
5) Reusability
6) Performance

Portability

It was our goal to create a software system that would be functional on as many different computer platforms as possible. A platform includes both the type of processor, and the operating system. More specifically, we wanted to develop a system that would run on all *NIX variants, Intel x86/Microsoft Windows, and Macintosh (note that MacOS X was not available in the beginning of the project, but its release will greatly simplify our eventual goal). This restriction alone severely

limits the choice of computer languages one can use. There are compilers for the C language available on just about every platform in existence. Additionally, Perl is also available on an incredibly large number of systems [33]. Unfortunately, neither of these languages supports any kind of graphical user interface, or image rendering directly. The original project design did not include plans for a user interface, or molecular rendering, but later development obviated the need for these tools. We settled on Tk for graphical user interface development, and OpenGL for graphics rendering, since both are available for both *NIX systems, and Microsoft Windows.

Scalability

While our own requirements for the developed software are quite moderate, we wanted to build a system that would eventually be useable on much larger problems. A typical liquid crystal molecule could weigh as much as 1000 atomic mass units or 160 atoms, but proteins can weigh much more, as many as 300,000 atomic mass units, or 50,000 atoms. Further, if one eventually used our libraries for simulation (a secondary consideration in development), one might need to have many large proteins resident in RAM at the same time. There are two specific areas of the program that are most effected by this issue.

Firstly, in the portions of the program written in C, we have created a 'fundamental atom type'. This is the data structure used to represent an atom, for any task. In the current implementation, 50,000 atoms require only about 15 megabytes of RAM, allowing for very large systems to be held completely within RAM.

Secondly, part of the program suite involves interface with a database of quantum chemical calculations. The database currently has around twenty entries,

and a typical entry on a Pentium III class system would take approximately 4 computer days to generate (note that these times are highly variable). When the database has grown to 2000 entries, the current program that serves information from it will have grown to 330 megabytes of RAM, once again, a somewhat moderate requirement for such a large amount of data.

Finally, the entire current code base is completely leak free (in terms of memory usage, and utility functions for freeing the more complex data structures are provided for ease of use by developers). Some libraries can be 'abused' in such a way as to introduce memory leaks, but this is unavoidable in a procedural (non object oriented) language such as C.

Usability

Normally when software is developed, only the end user is considered. Since it was clear from the outset of the project that the work could never be completely finished within the timeframe of a single thesis, it was decided that both the developer and the end user be strongly considered in the overall design and implementation.

One of the most difficult tasks for a programmer working (for the first time) on somebody else's code is to understand both the problem the original developer was trying to solve, and how they actually solved the problem. We have attended to this difficulty in four ways. Firstly, all source code is copiously commented (approximately 25% of the lines are comments). Secondly, we have broken the overall problem down into small enough steps that it should be reasonably easy to understand what the problem is, and in turn, how the portion of code solves that problem. Thirdly, since one of the easiest ways to understand a problem is by

watching the data flow through it, we have made all input and output be in text only format, and (hopefully) in plain English. Finally, by using procedural languages (C and Perl), we are forced to solve various problems in the same way that scientist generally do. There is much heated debate over what kind of language is better, but in our experience, scientists learn to solve problems by breaking them down, and taking steps, which is much more compatible with procedural languages than it is with object oriented languages.

Maintainability

This focus addresses not only maintenance, but extensibility as well. Extensibility is the process of adding onto existing work, without generating additional problems. We have addressed this issue in a variety of ways. As mentioned previously, all of the code is commented thoroughly, so it's easy for new developers to understand precisely what a given program or library does before they start work on it. We have also made every effort to separate the problems into 'specific' solutions, and 'general' solutions. This means that any code generated to deal with general solutions should be easily re-useable to solve other problems. It also makes the specific solutions more easily understood. Finally, consistently applied code formatting, long (descriptive) naming of variables and functions, and data abstraction that approximates chemists' notions all aid in the maintenance and extension of FFDev.

Reusability

This was largely addressed in the maintainability section. Reusability is the ability to take code that has already been generated, and use it elsewhere. The largest effort in this specific area was applied to the development of the atom_handling library, which was designed to do anything with the fundamental atom type that one might want to do. Where functions needed to be used by both C and Perl programs, libraries were written so that the same function would be available from both languages. Towards the end of the project, after I got a bit more experience with Perl, several reusable libraries were developed, to aid future development using that portion of the code base.

Performance

Performance is listed last in the list of major concerns for a very simple reason. All too often, pure focus on performance issues compromises all of the other important issues, as addressed in the previous sections. Performance has not been utterly neglected, however. It is widely accepted that compiled C code is the fastest form of executable, save for assembly or machine programs (which are utterly non-portable). Contrary to popular belief, however, Perl is not nearly as slow as many believe [34]. Perl is a (run time) compiled language (not unlike C), and the development time is much faster, since the programmer need not spend their time with memory management, or character by character manipulation. For these reasons, portions of the code that have heavy performance requirements have been developed in C, and the rest was developed in Perl.

Conventions

For all of the following sections, program names will be given relative to the 'ff' directory. After the first mention of a program, the program extension and/or directory prefix may be omitted. When examples with syntax are presented, items in angle brackets (<, >) are mandatory, and must be supplied verbatim, and options in square brackets ([,]) are optional. Items separated by the pipe symbol (|) represent valid options, but only one of the options may be specified (exclusive or).

On the topic of descriptors

For the average organic chemist, it's trivial to look at two atoms in two different molecules, and decide whether or not they're 'similar'. Those involved in generating force fields do this regularly, but our goal was different. In order to automate this comparison, we needed some way to assign real values that could be compared to the atoms in any given molecule. These values (almost certainly) must be numeric, and they must also somehow capture the essence of the 'character' of the atom in question. Careful analysis of how an organic chemist makes this comparison reveals that they must rely very strongly on two factors. Most importantly, chemists' notice what 'kind' of atom they're looking at (e.g., carbon, hydrogen, nitrogen, etc.). Secondly, they notice the bonding in the nearby environment; for example; is this atom aromatic? aliphatic? What is the hybridization? There are many algorithms available for detailing the notion of 'similariy' in organic chemistry; the one that suited our purposes was the qcode..

The ubiquitous qcode

Before any real discussion about the software can continue, one needs to have a solid understanding of what a qcode is (a way to generate atom types), and how we use them. Any typical atomistic simulation software requires that each individual atom have a particular 'type', which identifies it as somehow 'chemically different' from atoms of other types. In *ab initio* quantum mechanical electronic structure calculations, however, the closest concept to atom type is the type of the nucleus, which is required to know how much positive charge it has, yet provides no differentiation between different carbons, for example. This difference introduces an important problem in mapping *ab initio* data onto classical interaction potentials.

The typical solution in other force fields is to look at the atom in question, and 'categorize' it as one of the available atom types (i.e., an aromatic carbon might be C_R). This approach can be automated [35], and often is, but was unsatisfactory for our purposes, as it results in a huge loss of information garnered from our *ab initio* calculations. The solution to this problem lies in Edgardo Garciá's qcode algorithm [36].

A qcode is a vector (or list) of numbers that uniquely identifies an atom, based on its topology and the electronegativity of topologically relevant atoms. For the current project, qcodes of depth 20 (QDEPTH) have been used throughout, but other QDEPTHs would be easy to implement.

The following section is (unfortunately) quite techincal in nature, as it gives the specific algorithm for derivation of the qcodes. The algorithm for determining the qcodes of all of the atoms in a molecule is as follows:

1) Assign a reduced electronegativity to all atoms in the molecule, which is given by: $\sqrt{\frac{\text{Pauling electronegativity}}{1+\text{the number of bonded atoms}}}$, this is the 0th element of the qcode vector

2) From n = 1 to (QDEPTH -1), and for all atoms in the molecule, do the following. The qcode at the nth position is given by: $\left(\frac{\sum \text{Neighbor's qcode}[n-1]}{\#\text{ of Neighbors}} + \text{This atoms qcode}[0]\right) \Big/ 2$

3) For each atom in the molecule record the current value of qcode[0]. It is used in the next step.

4) In the last step, we convert the intermediate qcodes to final qcodes. From n = 1 to (QDEPTH -1), and for all atoms in the molecule, do the following. The qcode at the nth position is given by: $\frac{\text{qcode}[n]-\text{qcode}[0]}{\text{qcode}[0]}$

While the details of the algorithm are difficult to grasp, the results are absolutely ubiquitous to our work. Having a qcode available for each atom, however, is not enough for doing comparisons, we needed a way to be able to say whether two atoms are ‘similar enough’ (something very commonly done among chemists, but a bit difficult to implement algorithmically). We defined a ‘qcode deviance’, which compares two qcodes (many valued lists of numbers), and returns a simple scalar (one value, in this case, a number) that defines how ‘similar’ two qcodes (and thus, the

underlying atoms) are. In practice, the deviance takes the form of a floating point number, such as 3.185. The integral part of that number (3) indicates to what range the topology of the two atoms in question are identical (**Figure 10**). The fractional part (.185) roughly corresponds to a percentile rank of 'how close' the neighbors beyond the exact match range are. A low value would indicate that beyond the exact match range, the molecules are radically different. A high value would indicate that beyond the exact match range, the molecule retain a fair degree of similarity. Once again, the following discussion is anything but easy to read. Unless you're interested in the exact implementation, it may be irrelevant. This algorithm is implemented as follows:

1) Define a floating point tolerance. If the absolute value of the difference between two numbers is less than this value, the two numbers are considered identical. This is necessary, since all floating point numbers on any machine are inaccurate in the last decimal place.

2) For each value in the two qcodes (denoted hereafter as qcode1[n] and qcode2[n]) from n = 0 until then end of the qcode, compare the two values. If they are identical, move on.

3) For the last pair of identical qcode elements, record the exact match, which is n + 1 (since the first element of the qcode is numbered 0).

4) Define a weighting factor (0.5), a sum accumulator (0), and the exact match (found in step 3). For n = one past the last match to n

= the end of the qcode, add to the sum:

$$\left(e^{-weighting_factor \cdot (n - exact_match + 1) \cdot |qcode1[n] - qcode2[n]|}\right)^2$$

5) Since we want an average deviance, we set sum = sum / (qcode length – exact match). The sum now currently represents an 'average error' in the non-exact matching portions of the qcode. In order for it to display the proper behavior (i.e., small error give a large value in the fractional returned part), we need to do further manipulations.

6) Set our fractional match to $-\log\left(\sqrt{\text{sum}}\right)$ (chemists may recognize this as a variant of the p function). If the sum is 0, or our fractional match is less than 0, return (exact match + 0.999), since this is more or less a perfect fit past the exact match part. If not, our fractional match is set to $fractional_match / 25$, which casts it into the (approximate) range of 0.01 to 0.70. Return (exact match + fractional match).

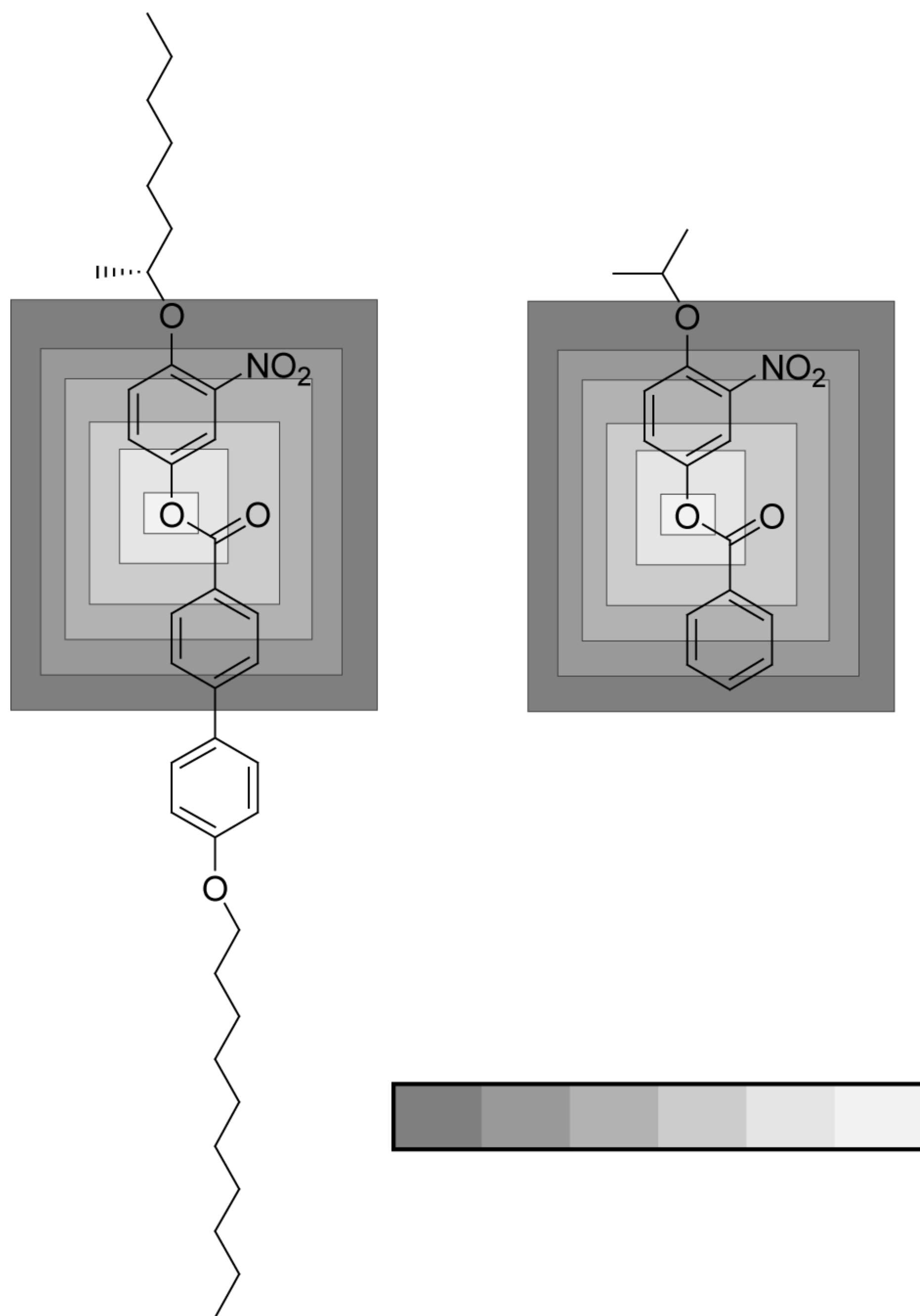


Figure 10: The "sphere of influence". For each of the atoms in the parent in all but the outermost shell, the corresponding fragment atom is connected to exactly the same atoms, and would give the appropriate "exact match".

The qcode deviance is then available to all programs written in C and in Perl (via an XSUB [37]). Empirically, this comparison gave deviances which agreed with 'chemists' intuition' in all but a handful of cases, out of 80. Out of the dubious

matches, all were ‘close calls’. Within the project overall, we frequently compare atoms, in which case, we use the above described deviance by itself. We also frequently need to compare ‘bonds’, which are identified by the atoms on either end. In that case, we use the geometric mean $\left(\sqrt{\text{deviance1} \cdot \text{deviance2}}\right)$.

Since qcodes only contain topological information, any stereochemical information is lost. To alleviate this problem, we defined our own scheme for assigning absolute configurations to tetrahedral stereogenic carbons. The algorithm is very similar to the CIP scheme [38], but instead of using that scheme’s prioritization, we relied on the qcodes to provide it. Using qcodes for this purpose makes the assignment more stable to small changes in connectivity, which was critical for our mapping fragment atoms and bonds to parent atoms and bonds.

Functional overview

This section will present a functional overview of how FFDev works. In very large software projects, it’s impossible to describe the entire system in one view. Regardless, we will attempt to present the overall operation and functions of the individual components in a single pass. This presentation is <u>very</u> loosely based on ideas from the Universal Modeling Language [39].

Collaboration Summary

According to “The Unified Modeling Language User Guide”, a collaboration diagram is an interaction diagram that emphasizes the structure organization of the objects that send and receive messages. Since our package is not written in and object oriented language, this view is not strictly applicable, but useful nonetheless.

Note that I have taken the liberty of using older symbols for presenting the behavior in this diagram, as they should be more familiar to readers.

Figure 11 shows the interactions of the major components of the software system. The quantum database portion (QDB) is an independent agent, and runs constantly. The generation system uses (and relies heavily upon) the QDB, and is designed to run through exactly once for any desired force field.

The QDB

The QDB consists primarily of four different programs (and several other very

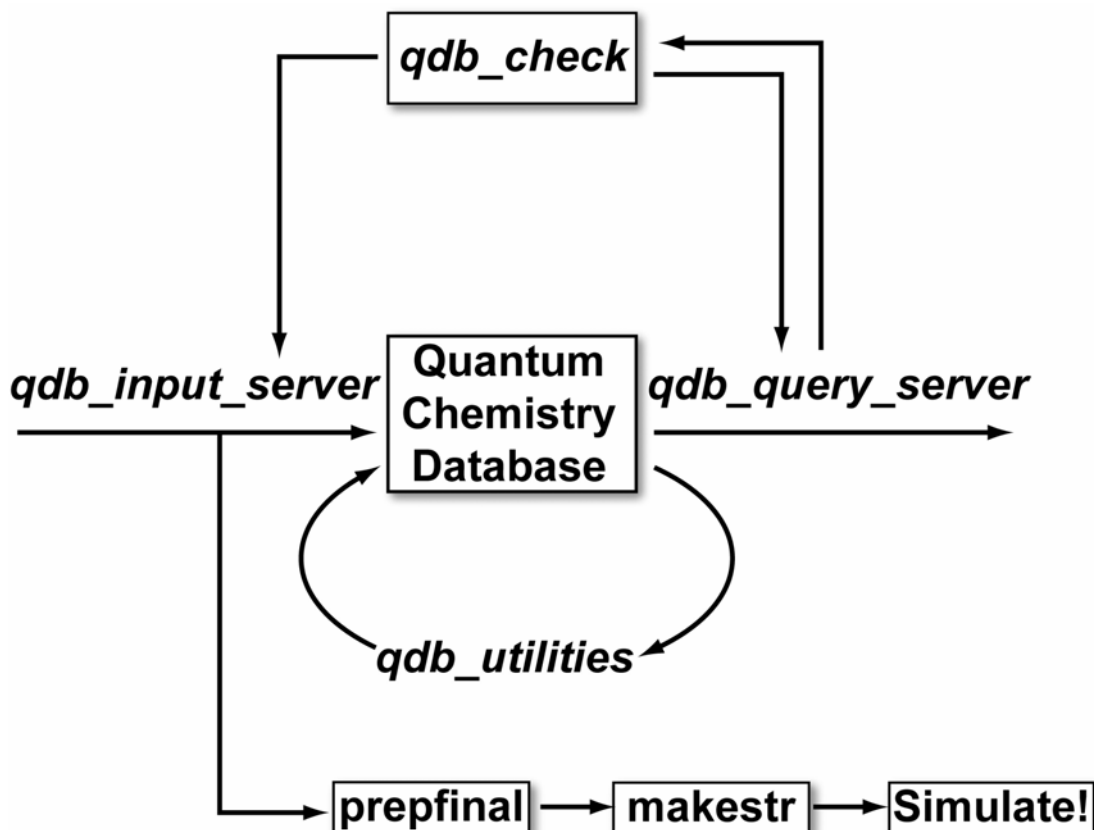


Figure 11: Collaboration summary diagram for the software. The central items represent the qdb agent, while qdb_check, prepfinal, and makestr represent the process agent.

small programs). These are: qdb/qdb_query_server.pl, qdb/qdb_input_server.pl, qdb/qdb_local_submit.pl, qdb/torsion_driver.pl and qdb/qdb_maintenance_utilities/qdb_utilities.pl. An overview of each of their functional behavior and responsibilities follows.

The program qdb_query_server is responsible for providing all database output. It is a daemon (runs constantly) that listens to a TCP/IP port on the host machine, receives plain text queries (one can use a telnet client to connect to it if they like), does a search, and returns the requested information. The commands it understand are as follows:

<get> [# of matches] <atom|bond> <match> {qcode_1_list} [{qcode_2_list}]

<get> <charge> <charge_type> <directory> <atom_number>

When making a query to the database, the user has the option of asking for several matches, or omitting the number of matches, and getting the 'best' one. Note that there may be several 'equally good' matches, and these will all be returned if that is the case. When requesting an atom match, the user should provide one and only one qcode. When requesting a bond match, the user must provide two qcodes. Note that in this case, the curly brackets have no special meaning typographically, but are required by the server to parse the qcodes.

The charge query is also quite flexible. Note that in order to request a charge, the client must know which atom on a specific fragment they want a charge for. This query finds all topologically equivalent atoms, and returns the average charge to the client.

The program qdb_input_server is responsible for placing all new *ab initio* calculations in the database, and requesting that the calculations be done. It is also responsible for starting all relevant torsion_drivers (which will be discussed shortly). In its current state, it is not a daemon, but a run-once type of program. When input is put into its input directory, and it is run, it creates and submits any new fragments to the local *ab initio* program and queuing system. It also starts torsions (via torsion_driver) that the input may have requested. The user is personally responsible for looking within the database for completed fragments (in control/qis/in_progress), and placing them in the root database directory. It would also be wise to re-run the request, as any necessary torsions belonging to new fragments are not calculated until qdb_input_server sees the relevant fragment in the database.

The program qdb_local_submit is responsible for managing jobs within the local computing environment. It is likely that if the entire system is ported to another computer (or group of computers) that this program would need to be heavily modified. This program takes requests for jobs in the control/que file, and when it's running the jobs, places them in submitted_jobs. It also leaves messages for the requesting processes (via a message.<pid> file), so they can continue their work, if they were waiting for the calculation to finish before proceeding. Before it actually submits a job, qdb_local_submit does its best to be the most polite user of the DEC cluster. It first starts by counting the total number of jobs the user is running. If it is above some maximum value, it refuses to submit the job. It then looks at each individual machine. If the requesting user has a job on that machine, it eliminates that machine from the potential candidates. It then tries to allocate approximately 1.8

times as much memory as the job is likely to take on each of the candidate machines. If any machine fails the memory allocation test, it is also removed from the potential candidates. Finally, it checks the load on each of the remaining machines three times, over the course of three minutes. The machine with the lowest load is selected to run the job.

The program torsion_driver.pl is responsible for running all of the torsions about any requested bond. It is started (automatically) by qdb_input_server.pl with information on which fragment, and which bond within that fragment we need angle vs. energy data for. If a directory already exists for the requested information (either because the previous torsion didn't finish, or perhaps because it is finished) it tries to restart any within that directory. In any case, what it does is run some number of torsions (provided from a configuration file) that are below some cutoff energy (again, from a configuration file, we've been using 20 kcal/mol as the cutoff). If there are no very high energy conformations (as would be the case for a torsion within an aromatic ring, for example), it will give some number of evenly spaced torsions. If there is a cutoff because of a high energy conformation, it will try to fill in as many angles as it needs to generate the requested number of data points. When it finishes, it offsets all of the energies so that the lowest energy conformation is at 0 kcal/mol, and records the information.

The maintenance of a (growing) database quickly became a concern. Each database entry has information about any stereochemistry the fragment may have, as well as other information. It became quickly apparent that we needed some way to check and repair the database as it grew. This is where qdb_utilities comes in. The

program has three modes (specified on the command line), namely summarize, verify, and update (which may have been better named ‘repair’). There is a subdirectory in its home directory called utilities, where small ‘helper’ programs reside. These include programs that re-generate qcodes, determine which bonds should be frozen in a torsion drive, etc. The program is designed so that it should be relatively easy to define a new task, and place the defining program in the utilities directory. It’s then only a few small edits to include the new test in qdb_utilities. After placing a new entry into the database (from a completed request by qdb_input_server), it is critically important to update all portions of the database, as the input server does not properly ‘condition’ the fragment.

The generation system

The generation system is comprised of a surprisingly fewer number of components. The program qdb/qdb_check is responsible for taking an initial structure and creating the initial file it needs for specification of the pending force field. The program cmap/map_charges.pl is responsible for mapping (and normalizing) symmetrized charges onto the parent molecule. The program finstr/prepfinalff.pl is responsible for gathering all of the information necessary to construct an arbitrary force field for any modeling package, and saving it in an easy to ‘reconstitute’ way. Formally, this is where the work of FFDev ends, but there is one more component that we need in order to create input for Matthew Glaser’s [7] modeling software. That program is finstr/makestr.pl. All of these programs are described in some detail in the following sections.

The program qdb_check is the does the core work of the force field preparation. Firstly, it takes the molecular structure (as an XYZ file, with or without connectivity information), and verifies that it can fully represent the molecule in its own native format. The checks include checking bond orders, formal charges, valences, and other relevant properties of the atoms. It then generates two lists. The first list is a list of all of the atoms in the molecule. The second list is a list of all of the bonds in the molecule. Here is where it begins its search for fragments to use to generate the final force field from. It establishes an internet TCP/IP socket connection with qdb_query_server, and 'asks' the server if it has relevant matches for each of the atoms and bonds that it needs. It records this information, and gets to work on atoms and bonds that the database has no relevant fragments for. For each of these 'orphan' atoms or bonds, it begins to grow fragments that would satisfy the 'similarity' criterion. The new fragments are real substructures of the original molecule, with hydrogen's provided as need to fulfill valence. It was in this phase that we really got to compare how well the qcode matching criterion worked, and the data is presented in **Figure 12**. For the three test cases, the fragments that match, as well as all of the atom and bond mapping for compound 1, can be seen in appendix B. Finally, a request is output, which is destined for qdb_input_server. The request is in the form of a file which lists the parent molecule, an atom in parent to atom in fragment (from the database, or new fragment) list, a bond in parent to bond in fragment list, and finally, a trailing list of any new fragments it would like added to the database.

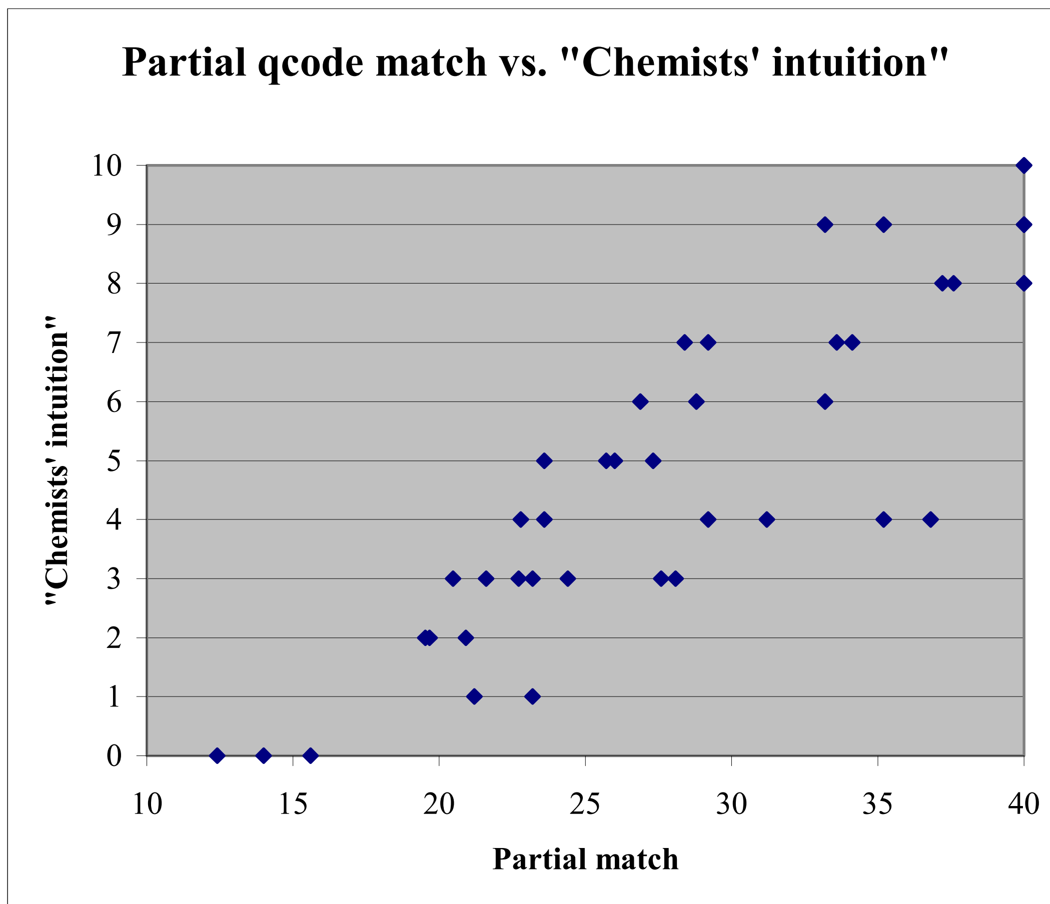


Figure 12: In the above graph, the exact partial match (times 100) on the x axis, and a 'chemist's intuition' as to how good the match is on the y axis. For partial matches over 40, the values have been changed to 40, which is considered a practical maximum (there were several such values in the dataset). The matches were atom to atom matches for a variety of fragments generated from the base structure of Compound 1. Note also that the exact match is not shown. The following numerical conversions for Chesmits' intuition were used: 0 is a very bad match, 4 is very acceptable, and 10 is perfect.

Unfortunately, the next step is to wait, since *ab initio* calculations can be quite time intensive. After all of the necessary calculations are done (or perhaps immediately, if 'good' fragments for everything in the parent molecule were already found), prep_final takes over. This program reads the (now mangled by qdb_input_server) file originally provided by qdb_check, and organizes all of the data into Perl data types. During this process, it runs map_charges, which simply queries qdb_query_server for symmetrized charges for all of the atoms, and then normalizes

all of the charges so the sum of charges on the parent molecule is 0. After prep_final is done, it dumps its initialized data into a file that is trivial to reconstitute in another Perl program.

It's easy to create 'all the information needed for any force field', but it's a much more difficult task to translate the information into useful input for some simulation package. This is where makestr comes in. As is the case with all modeling software that we are aware of, it is required that all atoms have 'types' associated with them. In this case, we assign somewhat arbitrary types to the atoms, such that only atoms with identical qcodes end up with the same label. The rest of the program assigns parameters (as described in Chapter 2) to all but the torsions, and maps bond lengths and angles from the *ab initio* minimized fragments onto the parent molecule. It then provides the user with a number of options for fitting the remaining torsions. The end result is a directory structure full of the relevant parameters, and a final master force field and structure, ready for input into a binding site calculation.

Other libraries and utilities

Aside from the main programs, there are a number of other programs that exist either to make life easier, patch known problems in the current implementation, or perform some other miscellaneous tasks.

In the root directory of the project, there's a program called Compile_all_fudge_scripts.pl. This program began as a 'quick way' for me to compile all of the C code in the project, but it has evolved into a multi-platform makefile maker, and project wide compiler. Running this (on your local machine) should compile all of the C code in the entire project, as well as the XSUBs needed

by the Perl programs that use get_qcode_deviance(). The project currently compiles effortlessly on Linux/PPC, Linux/ix86, and DEC alpha/OSF4 machines. System specific hints and configurations can be found in general/os_specific. On a related note, for every directory that has a compileable C program, there is a configure.pl file. This file will make a Makefile in the current directory, with all appropriate system specific options. It can also compile the program in a variety of ways, to support debugging, profiling, etc. Type “configure.pl –h” in any of these directories to see what options for configuring your Makefile are available.

There are several other files in the root directory of note. COMPATABILITY discusses any decisions that have been made that may affect portability. It also mentions any special libraries that the user may need to install on their system to be able to compile/use the package.

To_do.txt is full of exactly what it says. It notes any current limitations in various parts of the software. Some of the tasks may have been completed, and if they have been, it should be noted here.

The general subdirectory contains a variety of other ‘generally useful’ libraries. The program chkmem is a utility that is useful for determining if a machine is capable of allocating a given quantity of RAM, and is used by qdb_local_submit before submitting queries.

The core of our chemistry paradigm for the C code in the current project is encapsulated in the atom.h and atom_handling.c files, which together, represent our atom handling library. These files define, and provide functions for manipulating, our atom data type.

vector.c and vector.h are very basic (and quite inefficient) libraries for handling simple vector access and manipulation. They also provide very rudimentary support for some linear algebra functions.

total_atom_byte_size.c is a small utility that will tell you how big a single atom type in memory is. It can be used to make estimates of the size of large scale programs that use this atom type.

my_socket.c is a library for using internet TCP/IP sockets. It simplifies their usage, and gives some utility for receiving data, which is normally quite tedious, due to buffering considerations.

rc_file_handling.pl is an old style Perl library for getting options from the resource files used in this project. The only current files we use of this sort is in qdb/.qdb_checkrc, which has all of the configuration options used by various programs in the package.

clean_environment.pl is a library for un-tainting environment variables. Perl has a mechanism that allows the user to know when a variable may have come from an 'unsafe' source. If the relevant option is selected, Perl will not allow tainted variable to be used to output anything to the system. Occasionally, we need environment variables, and sometimes we need them in program for which internet security is a very important part of the programs function. In these cases, we manually un-taint the variables, and each instance of this is commented, with a risk assessment.

The doc subdirectory provides a (very old) Overview of the project, and a short tutorial on how to use CVS on the DEC cluster.

The genff and sim subdirectories contain the stub of a library whose original intent was to provide the classical energy evaluation necessary for the torsion fitting of the program. The relevant files are .ff_form, which is a plain text file describing the form of the force field desired, and nrgforce.c and nrgforce.h. The nrgforce library is designed to provide seamless integration with the atom_handling library, but more importantly, it is capable of run-time force field configuration, and hides nearly all details of other (proposed) functionality from the calling program.

The graveyard directory contains programs which have been abandoned in favor of redesigned programs. It may (or may not) contain code that could be useful for further development, but should not be used routinely in the program's normal operation.

The log2str directory contains just a bit of previous work not done by myself (log2str converts a log file to a str file). It also contains one function that is frequently used, called get_bond_order.c. This function assigns a bond order based on the atom labels, and the distance between them.

The one_timers directory contains programs that needed to be written to do one time functions (primarily database management), but there is no long-term need for their reuse. Once again, they may contain useful code to help meet future needs of the project, so they have been saved.

The perl_modules directory contains the Perl equivalent of C libraries. It is unfortunate that a couple of the modules have also ended up in general, but moving them to this directory would 'break' some existing programs. LINALG.pm is a module for performing linear algebra with native Perl data types. The most important

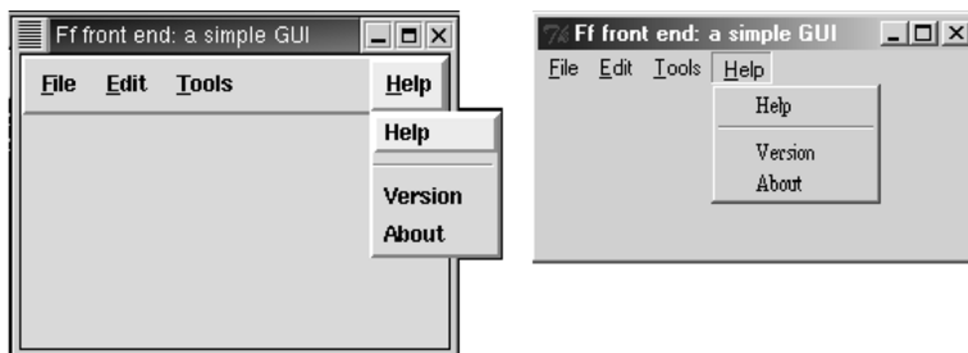


Figure 13: The program fffront.pl as it appears in GNOME (left) and Windows (right). In all x-windows implementations, the Help menu is supposed to be on the right side of the menubar, there is no such convention for Windows.

capability of the library is that it provides a simple way to get dihedrals angles in accordance with the standard chemist's convention [37]. NETFLOCK.pm is a module to provide file locking over NFS networks. It is a voluntary locking scheme, which means that in order for the locking to work, all programs that use a given file must use the same library.

The original design of the program required that the software not be dependant on the computing environment. This means that the user should not be required to use the same commercial *ab initio* program as we do, nor should they be required to have the same job queuing system as we do. Two Perl modules were written to serve this purpose. g98_functions provides easy ways to interface with Gaussian 98's input and output, without requiring the calling program to 'know' which library it's using. If a user has another *ab initio* program, they can simply copy this library, and re-write the functions to duplicate the behavior of the original ones. Similarly, local_functions provides an interface to behaviors specific to the users computing platform.

The qdb directory has seen the majority of development, and as such, has a variety of utilities that are not a part of the core implementation. format_for_g98.pl can be used to create .com files (for viewing with an appropriate molecular renderer) from .raw files, the format used by the database. kickstart_torsion_drivers.pl is a 'patch' program, to restart all of the torsion drivers, after killing qdb_local_submit. With some re-writing, this program will become obsolete. If you run this program, you will need to restart qdb_local_submit before the jobs will be resubmitted. As mentioned previously, the .qdb_checkrc file contains all of the configuration options that the various programs in the package use. format_connectivity.sh is a small utility to take a gaussian .com file with connectivity information, and create a corresponding file with connectivity in the style that our current simulation code uses. kqueryserver.sh will kill qdb_query_server regardless of what host it's currently running on. reghosts.sh is a small utility one can use to assist in setting up their ssh environment (which the current implementation of all inter-machine transactions is highly dependent on).

The runff directory contains a couple of 'proof of concept' programs. None of the work in this directory is ready for 'production use', but it may serve as a foundation for further development. fffront.pl is the beginning of a program designed to provide a GUI for all portions of the code base. When finished, it should have a database manager, and a force field creation manager. It is written in such a way that it runs with very similar results on both *nix systems, and Microsoft Windows (**Figure 13**). molren.pl is our own molecular renderer, and should eventually be able to read and render almost any molecular structure format known. It currently handles

only our own format, but even at its current level of development, creates quite nice renderings (**Figure 14**).

The shlib directory contains all XSUBs used in the program. Currently, the only shared capability we depend on is get_qcode_deviance(), but as the C and Perl portions of the code grow more interdependent, other XSUBS may be written.

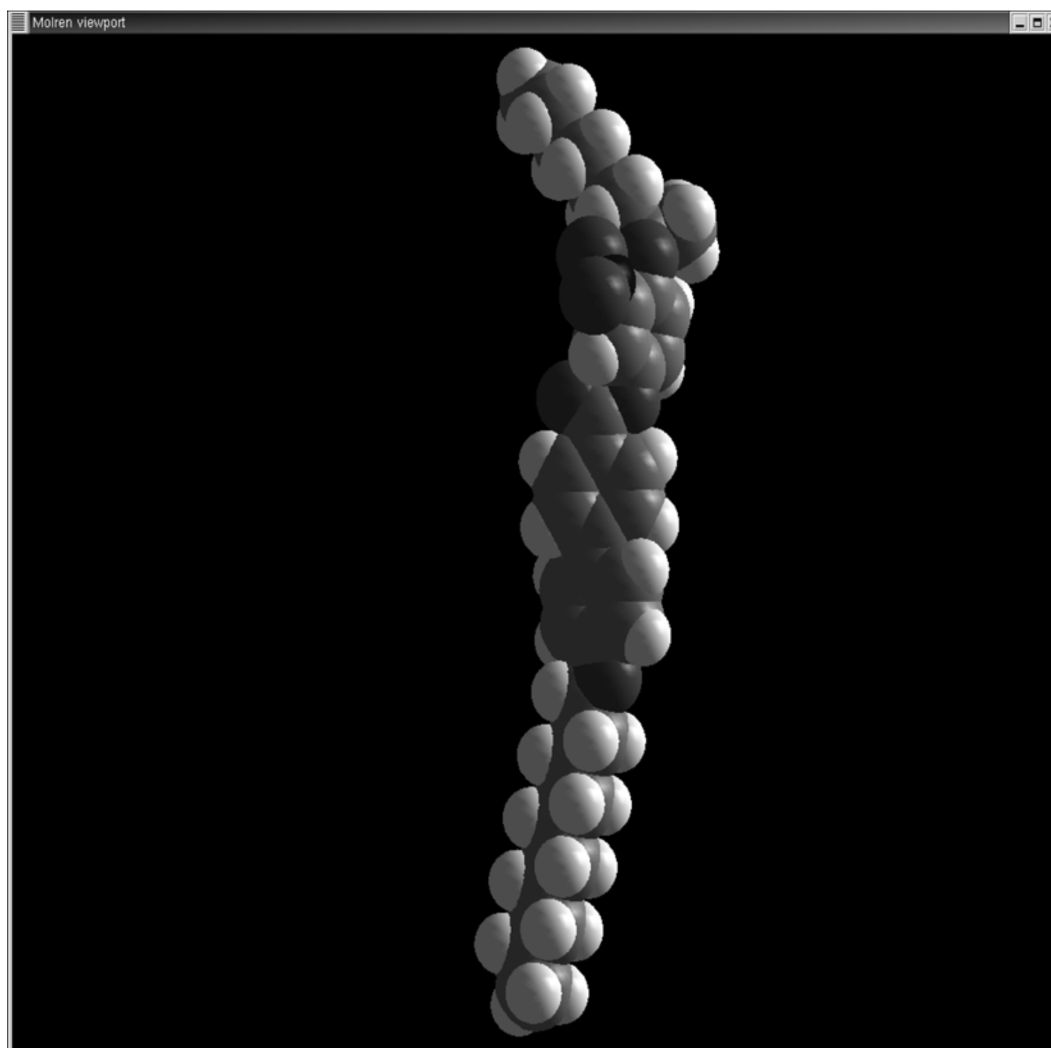


Figure 14: A rendering of Compound 1 from molren.pl.

Chapter 4

A tutorial

In this chapter we present a walkthrough of how the force field for our test compound (**Figure 7:** The compound for which force field was generated.) was generated. It will also cover 'variations' for procedures that are not encountered when generating this force field, but may be encountered for other compounds. It is intended to give the user of the software a template of how to do one of these, from beginning to end. Any data files that are generated by this run will be included in Appendix C.

Within this chapter, certain typographical conventions are used, to assist in clarity. These conventions were taken from "Programming Perl" [40]. *Italic* is used for path names, file names, and program names. `Constant width` is used in examples to show any literal output (or input) for programs, and relevant file contents. **`Constant width bold`** is used to indicate text that must be typed in exactly. *`Constant width italic`* is used to indicate that you must supply your own value. When there are optional values that you may have to supply, values in <angle brackets> represent mandatory values, while values in [square brackets] indicate optional values. If there are several valid choices <a|b|c>, they will be separated by the pipe character.

Two absolute paths will occur repeatedly in these examples, so we will shorten them. *qdb_path* is the path where your quantum chemistry database resides, in the case of the DEC cluster, this is */private_ffd/qdb*. The base path of

the program distribution will be indicated by *ff_path*. After changing to a directory, subsequent commands are assumed to have originated from the last directory used. The command prompt will be indicated by a `%` as the first character on the line.

Getting Started

Before doing anything, make certain that your own environment is set up completely. The package frequently needs to communicate between the various machines in the cluster. To verify you are setup correctly, type:

```
% /ff_path/qdb/reghosts.sh
```

This will attempt to log you into all of the machines in the cluster. If you have to provide a password, or type anything in, (but "exit", which you should type at each new login), then the software will not work until you have ssh set up properly. Setting up ssh is beyond the scope of this walkthrough.

Additionally, you will need to compile all of the C programs and libraries in the package. To do this, change your current directory to **./ff_path**, and run:

```
% Compile_all_fudge_scripts.pl
```

In order for the program to run, there are several daemons that need to be running. Begin by changing your current directory:

```
% cd /ff_path/qdb
```

Start the query server daemon. This daemon may be started anywhere, but it is imperative that it is run on the machine indicated by .qdb_checkrc within this directory. If you're not certain, open .qdb_checkrc with your favorite editor, and

find the line "`#query_server_host`". The next line is the host that the query server will be searched on. Note that you may need to make other edits to .qdb_checkrc to match your own computing environment. Start the daemon:

```
% qdb_query_server.pl &
```

The query server (as it is distributed with this thesis) may print out a lot of debugging information. This does not necessarily indicate that it's not working correctly, it just hasn't been removed yet. If you want to avoid having to see this, start the query server in another window, or simply redirect standard out to /dev/null.

Note that all of the daemons in the package are designed to catch SIGQUIT, and finish up gracefully. This is the preferred method for 'killing' the daemons. To find out what process id number (PID) the program is, type:

```
% ps -elf | grep qdb_query_server.pl | grep -v grep
```

You can then kill the appropriate program with:

```
% kill -SIGQUIT pid
```

If you are running this demo off of the enclosed CD (or an ISO image of the cd can be acquired from ffdev.sourceforge.net), all of the calculations already exist in the sample database, so no new *ab initio* calculations need to be run. If this is the case, please skip the next paragraph.

Now, we need to start up the local submission daemon. This daemon must be running on a machine that has access to the scratch directories of every machine, which also must be called /scratch_machinename. This is so the server can move the jobs to the correct machine before starting the Gaussian 98

calculations, to save on network communication. On the DEC cluster, this machine is jabberwock. Log into that machine, if necessary, before typing:

```
% qdb_local_submit.pl &
```

The path to patience

Now that we are ready to proceed, we'll begin with the fragmentation. Type the following to get started:

```
% qdb_check < samples/dave1.xyz > ff1.txt
```

```
Beginning 94 atom match queries.  Each dot represents 5
atoms.
                     \................./
                     \................./
Begin bond queries:  \................./
```

The second two progress bars will print periods as the program does its work. ff1.txt will contain much of the information necessary for the final force field, but may require further processing. Copy ff1.txt into the qdb_input_server directory as follows:

```
% cp ff1.txt /qdb_path/control/qis/input/ff1.txt
```

Once again, if you are using the sample database, you will have no need to run any *ab initio* calculations. In this case, you may skip the next step. If you run the input server when all of the information is already in the database, the server will do nothing but go to sleep, waiting for some input that would need to have calculations run on it. Run the input server:

```
% qdb_input_server.pl
```

The input server may make new entries into the database, or run one or more torsion_driver.pl daemons. One can check the database for unfinished torsions by typing:

```
% chkincompletetorions
```

If there are incomplete torsions, they may or may not belong to your compound. The ff1.txt file we generated is human readable, so the curious can look through it to see which torsions on which fragments will be required to parameterize torsions within the parent molecule. Conceptually, there are only two types of entities that need to be mapped from fragments onto the parent. These are atoms, and bonds. Three and four body interactions all have either an atom, or a bond, that they are centered on. A typical line (from the bonds section) looks like this:

```
Dir: C20H16O3-0 Parent bond 10-15: Qdb bond 10-15: qdb homo
```

The Dir section is the name of the database entry for the fragment that will be used for this particular bond. The numbering for the bonds are all zero based, which means, depending on your method of visualization, you may need to add one to the atom numbers to get the correct bond (gaussview uses a 1 based numbering system). The 'qdb' at the end of the line indicates that the fragment exists in the database when the program was run, it may say 'frag <#>', if an appropriate fragment did not exist, in which case, the fragment specifications will appear at the end of the file. The homo (at the end) means that either there were no tetrahedral stereogenic carbons in the molecule, or that the parent molecule and the fragment molecule have the same absolutely configuration at all

tetrahedral stereogenic carbons. It could also be ‘enantio’, or ‘diastereo’. If it is enantio, the torsion vs. energy will have to be reflected, before being fitted, if its diastereo, the fragment would not be appropriate. Since none of the test cases would have this problem, the code to handle these variations is not currently developed, though they will trigger errors if detected.

If qdb_input_server made any new entries into the database, they will automatically be run by qdb_local_submit (which you started earlier). For a record of what jobs was submitted to what machine, and when, read /qdb_path/control/qdb_local_submit_err.log. In the case that there were new fragments submitted, they can be found in /qdb_path/control/qis/in_progress, under the name of the file you submitted, in this case, ff1.txt. Since the input server is not ‘finished’ yet, the user must wait for the selected calculations to finish, and manually copy any new fragments into the database, for example, with something like this:

```
% mv /qdb_path/control/qis/in_progress/ff1.txt/C2H4O-3 /qdb_path/
```

After adding any new fragments to the database, you must run the maintenance utility to ‘finish’ the database entry:

```
% cd qdb_maintenance_utilities
% qdb_utilities –ua
% cd ..
```

Hopefully, you will have not had to wait too long for the *ab initio* calculations to finish, or better yet, perhaps all of the entries are already in the database!

Completion

All of the *ab initio* calculations are finished, and you're ready to complete your force field. Before completing the force field, you need to make sure that qdb_query_server.pl is running, see the previous section for information on how to start (and stop) this daemon. At this point, you need to regenerate the ff1.txt file with qdb_check (unless qdb_input_server didn't have to start any new jobs for you). Follow the above instructions to do so. In future implementations, it will be left (in a finished form) in the /qdb_path/control/qis/output directory, and you'll not need to regenerate it.

To generate the data necessary for completion of <u>any</u> force field, run:

```
% /ff_path/finstr/prepfinalff.pl ff1.txt > ff1.fff
```

ff1.fff (final force field) is a (barely) human readable file, which contains all the data necessary to complete a force field of any design. If you do not use Matthew Glaser's simulation/torsion fitting code, then this is the point that the software ends, for you. If you do choose to read it, read on!

Closure

While the generic force field is finished, there is nothing like a useable force field yet. This is very much dependent on what simulation software you'll be using. Here, I discuss the usage of Matthew Glaser's torsion fitting, and simulation code. In order to use some of the features of makestr.pl, you will need to have the programs build_single and minimize in your path. To continue, do something like in the following example:

% /ff_path/finstr/makestr.pl ./ff1.fff

```
o)  Overwrite the directory structure and initialize
r)  Refresh the directory structure without destroying existing files
s)  Skip all initialization, and go into interactive mode immediately
t)  Try to fit and verify all torsions, this option is dangerous, and
    will definately take some time.  It will also _not_ initialize the
    directory structure, so you should refresh or overwrite if you're
    not certain the directories are properly set up.
q)  Quit before doing anything
What shall we do? (o|r|s|t|q) [s] o
Initialization progress: \......../
                         \......../
Entering interactive mode:

1)  Change current fragment/torsion (C8H18O-0, 1)
2)  Delete all information for current fragment/torsion
3)  List all torsions with their status
4)  Fit current torsion
5)  Check log file from last fitting run
6)  Verify current torsion
7)  View graph of fit
8)  Declare this torsion finished
9)  Quit
Your choice? (1|2|3|4|5|6|7|8|9)  9
```

Now run it again:

% /ff_path/finstr/makestr.pl ./ff1.fff

```
o)  Overwrite the directory structure and initialize
r)  Refresh the directory structure without destroying existing files
s)  Skip all initialization, and go into interactive mode immediately
t)  Try to fit and verify all torsions, this option is dangerous, and
    will definately take some time.  It will also _not_ initialize the
    directory structure, so you should refresh or overwrite if you're
    not certain the directories are properly set up.
q)  Quit before doing anything
What shall we do? (o|r|s|t|q) [s] t
Note that even after running the torsions, you will need to manually
check them to make sure the fits are good, etc.  Feel free to simply
re-run this program after the batch is done, then select s (to skip
the directory initialization).  Also, be certain to enter the new
values into the master force field.
a)  Run all possible fits and verifies
u)  Run all unfinished (as marked in the master/completed_torsions
          file)
o)  Run only torsions for which there is no fit or verify directory
q)  Quit
Your choice? (a|u|o|q) a
User requested 160 tasks
```

This will begin the fitting and verification of all of the necessary torsions. This process will likely take at least fifteen minutes, and may take as long as a couple of hours, so feel free to take a break. After it finishes, take a look at the

graphs for each of the fits (simply re-run the program, skip initialization in the first step, and follow the menus). If there are problems with any of the fits, any other corrections would need to be done manually. When you are content with any given fit, select the ‘Declare this torsion finished’ option to enter the new parameters into the final force field. After all torsions have been entered, the final force field will be done, and ready for simulation. It will be in ./myff/master/master.mff, and the structure will be in ./myff/master/master.str. Note that a sample (completed) ./myff directory is included on the CD.

Happy simulating!

Chapter 5

What's New, revisited

While Chapter 1 mentioned many of the features of FFDev with respect to a light history and background of quantum chemistry, statistical mechanics, and computer simulation, a more thorough and succinct presentation of the novelty and usefulness of the current work is called for.

High quality force fields of arbitrary forms from first principles

There is a true plethora of force fields in existence [6], and available to the academic community. We propose, and have implemented, a procedure for the rapid generation of custom, appropriate, and disposable force fields from *ab initio* data. Since the generation of a single custom force field is routine, we expect to be able to quickly test a variety of forms and parameters, and allow other users to generate force fields most suitable for their own applications. A variety of other benefits arise from our approach.

Background

Despite the large number of force fields available to researchers today, our own requirements found them all lacking in some important area or another. Specifically, the types of simulations we do require that the potential energy of a molecule as a function of the various dihedrals be as precise and accurate as

possible. Many have made custom force fields for their own (specific) purposes, including ourselves [7]. The process of developing one's own force field, however, is fraught with difficulties.

Like many others before, we wished to use *ab initio* data as the basis of our force field, and to generate a classical expression that most closely reproduces the quantum chemical energy surface. In our experience, even a researcher skilled at generating custom force fields will require several weeks to several months to create a single force field. The procedure involves numerous transcriptions, and scores of objective decisions. Humans are all too error prone when it comes to transcription, and the sheer number of objective decisions that need to be made seems to defy recording and reporting (in a journal article, for example).

Our solution to the most obvious problems was to generate the force fields with software. This serves to both document the procedure we used, and to automate the creation of future force fields.

Motivation

Force fields are the foundation for any kind of simulation, and contain two parts; the form of the force field, and the parameters. The form is a function that gives the energy as a function of the positions of the members of the system. The parameters are the actual constants that are put into the form to give it the correct behavior.

Whenever a molecular simulation is run, there are three potential sources of error. Firstly, the model used for the simulation may not accurately represent

what's happening at the molecular level. Note that the model includes information about the method we use to run the simulation (molecular dynamics, Monte Carlo, etc.), as well as other simplifying assumptions, such as a mean field. Secondly, the form of the force field may not be capable of precisely representing the energies of the system. Thirdly, like the form, the parameters used in the force field may be at fault. Note that all three of these are intimately intertwined, and they cannot necessarily be separated from one another cleanly. Regardless, we make the distinction to try to understand the source of inaccuracies in simulation.

It is critical to note here that parameters for one form of a force field are <u>not</u> transferable to other forms. Unfortunately, all too often in simulation literature, this subtle fact is lost. Van Gunsteren [23] discusses this problem very thoroughly. In addition to what kind of terms are summed to give the total energy (such as bond stretching, etc.), the form also includes the following: Where there any cutoffs used in evaluating the columbic or van der Waals forces? What were the cutoffs? What type of cutoff was it? Were one-four interactions included, excluded, partially included? What combination rules were used for hetero-dispersion terms? Were the van der Waals forces evaluated with a Lennard-Jones potential, or an Exponential-6 potential?

To answer all of the previous questions, and the many that were omitted, one must be able to take a look at the program code used to evaluate the energy expression. In most applications, the form of the force field is 'hard wired' into the code.

Once the form of the force field is (completely) known, the parameters must be called into question. Why were the values chosen? What assumptions were made in selecting the values? There are so many questions of this nature that they can never be 'manually' enumerated in a publication of the force field.

It is our belief that the most accurate and objective source of data for parameterization of force fields is from *ab initio* calculations, which can be 'arbitrarily' exact. Once one can feel confident that the parameters for whatever form of force field they're using are 'as good as possible', simulation reveals shortcomings in either the model or form of the force field; the uncertainty about the parameters is gone. If generating force fields for arbitrary forms, and generating appropriate parameters for that form becomes routine, then rapid 'screening' of forms for a given model opens the door for rapid refinement of the form, until a suitable form for the problem at hand emerges.

Procedures and Justification

One of the fundamental requirements of all force fields is that the atoms be assigned a type, as the individual energy terms require the atoms to have some kind of name, or identity. Different research groups have come to widely divergent conclusions about how many atom types are 'necessary' to represent the range of chemical variability in a given molecule. Our solution to this (now long standing) argument was to allow <u>every</u> topologically and stereochemically inequivalent atom to have its own atom type (this is a slight misnomer, as enantiotopic atoms a certain distance from asymmetric stereogenic carbons are allowed to have the same atom type). This is done by using a descriptor scheme

(qcodes) developed by Edgardo Garciá [36], and our own stereogenic carbon descriptor scheme.

Since we wished to parameterize our force fields from *ab initio* data, we had to make the assumption that properties of atoms (or bonds) in a large molecule can be adequately represented by atoms or bonds from smaller molecules with similar electronic and topological structure. By generating our own qcode comparison metric, we can determine which smaller (and therefore amenable to quantum chemistry calculations) molecules would be suitable proxies for atoms and bonds in the larger molecule. We have dubbed the process of generating a list of small molecules necessary to represent a large molecule 'fragmentation'. During this process, we also generate a 'map', which indicates which fragment atoms and bonds will be 'stand-ins' for atoms and bonds on the larger molecule.

Ab initio calculations are very time consuming. Our prototype work has shown that we need to generate approximately 1/6 the number of fragments as there are atoms in the molecule. If we needed to do quantum chemical calculations on all of those fragments for every force field, our productivity would be severely limited by computer time. To alleviate this problem, we have developed a quantum chemistry database, as a way to archive previous calculations. This allows the data to be reused indefinitely. It also allows the form of the force field to change arbitrarily, since the underlying data remains accessible.

Unlike conventional force fields, we are not limited to a certain portion of the periodic table for which parameters have been determined. Any atom that can be used in an *ab initio* calculation can be used in one of our force fields.

In many ways, our approach may seem like overkill. We are able to refine our form and parameters until we come 'arbitrarily close' to exactly reproducing the *ab initio* potential energy surface. Conventional wisdom declares that, while the intra-molecular interactions may be important, the inter-molecular interactions completely dominate the bulk behavior. (For clarity, we use the common vernacular that considers bonded interactions to be intra-molecular, and non-bonded interactions to be inter-molecular; even though non-bonded interactions occur between atoms in the same molecule.) The topic of how to get inter-molecular interactions (columbic and van der Walls) from *ab initio* calculations is one of very active research right now, and we haven't begun to tackle it, instead opting to concentrate on the intra-molecular potential. Why such precision?

The simplest retort to this question is: Why not? We have found it relatively easy to get arbitrarily good intra-molecular parameters, and, though the uncertainty is much less than the uncertainty in inter-molecular parameters, the precision is available to keep up with future advancements. Additionally, some models use a mean field in a vacuum, and require <u>only</u> intra-molecular parameters; these types of simulations can benefit greatly from the additional precision.

By feeling confident that our intra-molecular potential is accurate, we can turn our attention to the inter-molecular portions of the force field. Since we can

tune how we get intra-molecular parameters from our database, and routinely generate new force fields, we are able to prototype, test, and refine our force fields rapidly. Polarizable charge models are gaining much popularity in the current literature. Parameterization of these models is nearly impossible to do from experiment, which means researchers must instead rely on quantum chemical calculations. **Atomic** (atom centered) polarizability is a fine concept (as are point charges on nuclei), but there exists no quantum mechanical operator for either, unless the entire molecule is a single atom. One can envision a great number of ways to do this, and we look forward to being able to join the current researchers in trying to solve this problem.

Conclusions

The ability to rapidly create many force fields of arbitrary form, from a well defined procedure, is a great boon to anybody interested in doing molecular simulation. It is well accepted that different force fields are 'better' at some kinds of simulations than others. Imagine rapidly generating twenty different kinds of force fields for a particular task, and evaluating the results of simulations using each of them. This would allow a person doing simulations to very quickly find the most appropriate force field for their current problem.

Many perceived shortcoming of existent force fields are resolved by generating a 'disposable' force field when you need it. Firstly, the entire procedure is fully documented (via the source code), and anybody can reproduce the results. Secondly, since generation of new force fields is routine, we are freed to concentrate our efforts on improving the form of our force fields. Thirdly, by

assigning a different atom type for every unique atom in a system, our force fields are both flexible, and ultimately appropriate for whatever the current task may be. Finally, we can reach ‘arbitrary’ precision, provided the property in question can be treated and solved in a quantum mechanical calculation.

Chapter 6

Wrapping it all up

In this chapter, we will wrap up all of the loose ends left during the previous chapters. Specifically, we will discuss where the software and other supplementary materials can be found, what our accomplishments are, and what science we hope to promote in the future based on this work.

Supplementary materials

The supplementary materials for the presented work are in digital format, and I have chosen two independent places to 'officially' archive it. Firstly, if you have an 'official' copy of this thesis, there will be an attached CD. The CD has four files in the root directory.

The 'ff' directory contains all of the code in the project, as well as compiled executables for a Linux 2.2x/686 kernel, though the code should be easy to recompile for your own system. There is an ff1.fff file, which is referenced in Chapter 4, and is an 'almost finished' force field. There is a truncated *ab ititio* database in the 'qdb' directory, as it is required by the demo. Finally, there is a 'myff' directory, which is created if you follow the last step in the tutorial, and have access to Matthew Glaser's simulation code.

All of the data necessary to follow the tutorial in Chapter 4 is on that CD. If you have come by this document by other means, you can find a gzipped ISO of the CD at ffdev.sourceforge.net. The 'thesis final' release of the software will be

available there, as well as any 'current' releases. ffdev.sourceforge.net will be the permanent home of the project, so if you are interested in contributing to the project, or know of someone that would be, please visit that site.

All of the software generated in the work leading up to this thesis is copyright Joshua Radke, 2002. It is openly available for any user, and is licensed under the Gnu General Public License [41]. This particular license was chosen to protect the future of this project as a community effort, and to allow it to live in perpetuity in the public domain.

Accomplishments

This work has made several ground breaking advances in the preparation of classical interaction potentials for atomistic simulation. First and foremost, we have demonstrated that it is possible to completely automate the process of taking a single (potentially large) molecule, and create from scratch (*ab initio* data) all of the data necessary to create a force field completely from first principles. We have further demonstrated the re-use of expensive *ab initio* quantum chemical calculations, and made the 'data mining' necessary for this task simple for the end user. These two tasks serve as a proof of concept that creation of <u>purely</u> *ab initio* force fields is possible.

By casting our force field into a form suitable for Boulder Model binding site simulations, we have shown two things. Firstly, we have demonstrated a practical application of the automated force field creation. Secondly, we have provided further evidence of the usefulness of the simple Boulder Model mean field approach for determining both the sign and magnitude of macroscopic polarization.

Finally, and perhaps most importantly, we have opened the door to a completely new approach to the refinement of force fields. The focus for improvement of a force field of a given form can now easily be treated as a problem of how we parameterize it (from fundamentally sound input), instead of the historical approach of tweaking parameters without justification.

Future work

We have by no means created the be all and end all of force field creation. In fact, perhaps our biggest accomplishment is in the number of new research directions we have created. As mentioned in Chapter 2, our force field is by no means derived strictly from *ab initio* data, though the final torsion fitting serves to sweep the inadequacies in the empirical parameters into the torsion terms. Several very interesting possibilities arise with our new methodology.

We have used incredibly generic bond stretching and angle bending parameters in our own force field. We consider this a reasonable approximation for our purposes, as they have little bearing on the overall shape of the molecule. In order to get classical force fields that are capable of reproducing infrared spectra of molecules, we would need much more sophisticated parameterization. Firstly, we would need to extract second derivatives of the energy with respect to nuclear motion for the relevant parameters. This is in fact data easily accessible in some kinds of *ab initio* calculations, so would fit very well into our data extraction approach. Secondly, we would need to add coupling terms, another task that lends itself well to extraction from *ab initio* calculation data.

Peter Tieleman, a membrane biophysicist at the University of Calgary told me several years ago: ”If you want to do solution phase simulations, quantum mechanics is practically useless …” While his statement may be true to an extent, we remain optimistic that the ‘real’ answer lies in understanding inter-molecular interactions at the quatum chemical level. To this end, we have several ideas for getting arbitrarily precise parameters for either the Lennard Jones potential we're currently using, or for parameterizing any other form of intermolecular potential. There is also currently work being done on doing *ab initio* calculations in ‘effective solvent fields’, though we feel that this (semi-empirical) approach suffers from the same limitations as other semi-empirical approaches. This is an area that would be very interesting to pursue in the future.

Finally, one of the most exciting new fields of work in force field development involves the usage of polarizable charge models. These models all allow the charges to either float off of the nuclei, or allowing the charge to redistribute itself within the same molecule. Regardless of the form of force field that we use, our methodology for mapping from small fragments onto large parent compounds should prove ubiquitous for this parameterization.

In closing …

Force field creation need not be an activity limited to the few experts in the world. What started as a simple request grew into a suite of programs suitable for the simple, rapid, on-demand creation of strictly appropriate force fields for arbitrarily large molecules. Admittedly, it is only a beginning; yet we believe our unique

approach, once fully realized, could revolutionize the way force fields are created, refined, and used today, and for the foreseeable future.

Bibliography

1) a) Walba, D. M., Slater, S. C., Thurmes, W. N., Clark, N. A., Handschy, M. A., Supon, F. J., *J. Am. Chem. Soc.*, **1986** 108, 5210; b)Walba, D. M., Vohra, R. T., Clark, N. A., Handschy, M.,A., Xue, J., Parmar, D. S., Lagerwall, S. T., Skarp, K., *J. Am. Chem. Soc.*, **1986**, 108, 7424; c) Walba, D. M., Clark, N. A., c) Walba, D. M., Clark, N. A., *Ferroelectrics*, **1988**, 84, 65, e) Walba, D. M., Razavi, H. A., Clark, N. A., Parmar, D. S., *J. Am. Chem. Soc.*, **1988**, 110, 8686.

2) a) Bartolino, R., Doucet, J., Durand, G., *Ann. Phys. (Paris)*, **1978**, 3, 389, b) Yoshizawa, A., Kikuzaki, H., Fukumasa, M., *Liq. Cryst.*, **1995**,18, 351.

3) a) Maier, W., Saupe, A. Z., *Naturf.*, **1958**, A13, 564, b) Maier, W., Saupe, A.Z., *Naturf.*, **1959**, A14, 882.

4) Allen, M. P., Tildesley, D. J. *Computer Simulation of Liquids*, Oxford Science Publications: Oxford, New York, 1994.

5) Meyer, R. B., “Structural Problems in Liquid Crystal Physics” in *Molecular Fluids*, Balian, R, Weill, G. , Eds.; Gordon and Breach, London, 1976. This paper presents a written account of work presented at the “Summer School of Theoretical Physics”, held in Les Houches, France, France, August, 1973. Meyer also presented this idea, along with preliminary experimental results, in a famous talk at the Vth International Liquid Crystal Conference in Stockholm, in 1974.

6) The field of molecular dynamics has exploded in recent years, and a great variety of software has been developed for either generating force fields for molecular dynamics simulations, or for actually running the simulations. There are many more than listed here, but a partial list includes: AMBER[a], BRUGEL[b], DEDAR[c], CHARMM[d], EGO[e] ENCAD[f], FOCUS[g], GROMACS[h], GROMOS[i], MOIL[j], NAMD[k], POLARIS[l], UHBD[m], X-PLOR[n] YASP[o], CHARMm AND DISCOVER[p], AND SYBYL[q]. a) Pearlman, D. A., Case, D. A., Caldwell, J. W., Ross, W. S., Cheatham III, T. E., DeBolt, S., Ferguson, D., Seibel, G., Kollman, P., *Comput. Phys. Commun*,**1995**, 91, 1, b) Delhaise, P., van Belle, D., Bardiaux, M., Alard, A., Hamers, P., van Cutsem, E., Wodak, S. J., *J. Mol. Graphics*, **1985**, 3, 116, c) Carson, M., Hermans, J., *Molecular Dynamics and Protein Structure*, Hermans, J., Ed.; University of North Carolina, Chapel Hill, 1985, pp. 165-166, d). Brooks, B. R, Bruccoleri, R. E., Olafson, B. D., States, D. J., Swaminathan, S., Karplus, M., *J. Comput. Chem.* , **1983**, 4, 187, e) Eichinger, M., Grubmüller, H., Heller, H., *User Manual for EGO-VIII, Release 1.0*, Universität München: München, 1995, f) Levitt, M., Hirshberg, M., Sharon, R., Daggett, V., *Comput. Phys. Commun.,* **1995**, 91, 215, g) Lemon, A. P., Dauber-Osguthorpe, P., Osguthorpe, D. J., *Comput. Phys. Commun.*, **1995**, 91, 97, h) Berendsen, H. J. C., van der Spoel, D., van Drunen, R., *Comput. Phys. Commun.*, **1995**, 91, 43 i) van Gunsteren, W. F., Billeter, S. R., Eising, A. A., Hünenberger, P. H., Krüger, P., Mark, A. E., Scott, W. R. P.,

Tironi, I. G., *Biomolecular simulation: The gromos96 Manual and User Guide*, Hochschulverlag an der ETH Zürich: Zürich, 1996, j) Eiber, R., Roitberg, A., Simmerling, C., Goldstein, R., Li, H., Verkhivker, G., Keasar, C., Zhang, J., Ulitsky, A., *Comput. Phys. Commun.*, **1995**, 91, 159, k) Nelson, M., Humphrey, W., Kufrin, R., Gursoy, A., Dalke, A., Kale, L., Skeel, R., Schulten, K., *Comput. Phys. Commun.*, **1995**, 91, 111, l) Lee, F. S., Chu, Z. T., Warshel, A., *J. Comput. Chem.*, **1995**, 14, 161 **1993** m) Madura, J. D., Briggs, J. M., Wade, R. C., Davis, M. E., Luty, B. A., Ilin, A., Antosiewicz, J., Gilson, M. K., Bagheri, B., Scott, L. R., McCammon, J. A., *Comput. Phys. Commun.*, **1995**, 91, 57, n) Brünger A. T., *X-PLOR: A System for X-ray Crystallography and NMR*, Howard Hughes Medical Institute and Yale University: New Haven, 1996, o) Müller-Plathe, F., *Comput. Phys. Commun.*, **1993**, 78, 77, p) Accelrys, Burlington, MA 01803, 1997, q) Tripos Inc., St. Louis, MO 63144, 2002.

7) a) Glaser, M. A., Clark, N. A., Garciá, E., Walba, D. M., *Spectrochimica Acta Part A*, **1997**, 53, 1325, b) Garciá, E., Glaser, M. A., Clark, N. A., Walba, D. M., *J. Mol. Struct. (Theochem)*, **1999**,464, 39, c) Glaser, M. A., Clark, N. A., submitted to *Liquid Crystals*, **2002**.

8) Tsaparlis, G., *Chemistry Education: Research and Practices in Europe*, **2001**, 2(3), 203.

9) Heisenberg, W.Z., *Zeitschrift der Physik*, **1925**, 33, 879.

10 Born M., Jordan, P., *Zeitschrift der Physik*, **1925** ,34, 858.

11) a) Schrödinger, E., *Annals der Physik*, **1926**, 79, 361, b) Schrödinger, E., *Annals der Physik*, **1926**, 79, 489, c) Schrödinger, E., *Annals der Physik*, **1926**, 79, 734.

12) Dirac, P. A. M., *Proc. Roy. Soc (London) A*, **1929**, 123, 714.

13) In 1998, John Pople won the nobel prize in Chemistry "for his development of computational methods in quantum chemistry." For a complete bibliography of John Pople's work from 1950 to 1989, see *Int. J. Quant. Chem.*, **1990**,38, 355-371.

14) No published reference can be found, but it is known that Gordon E. Moore (one of the co-founders of Intel) first made his statement in 1965.

15) Intel Corporation. Moore's Law. http://www.intel.com/research/silicon/mooreslaw.htm (accessed May 2002).

16) Biggus, J. Sketching the History of Statistical Mechanics and Thermodynamics (From about 1575 to 1980). http://history.hyperjeff.net/statmech.html (accessed May 2002).

17) Circa 150 BC, Hero of Alexandria, *Pneumatics*. For an english translation of this work, see Woodcroft, B. THE PNEUMATICS OF HERO OF ALEXANDRIA. http://www.history.rochester.edu/steam/hero/ (accessed May 2002).

18) Waterston, J. J., *Thoughts on the Mental Functions*, self-published, 1843.

19) a) Metropolis, N., Rosenbluth, A.W., Rosenbluth, M. N., Teller, A. H., Teller, E. *J. Chem. Phys.*, **1953**, 21, 1087, b) Rosunbluth, M. N. Rosenbluth, A. W., *J. Chem. Phys.*, **1954**, 21, 881, c) Alder, B. J. Wainwright, T. E., *J. Chem. Phys.*, **1959**, 31, 459.

20) Gibson, J. B., Godland, A. N., Milgram, M., Vineyard, G. H., *Phys. Rev.*, **1960**, 120, 1229.

21) Allinger, N. L., *J. Am. Chem. Soc.*, **1977**, 99, 8127.

22) a) Maple, J. R., Hwang, M.-J., Tosckfisch, T. P., Dinur, U., Waldman, M., Ewig, C. S., Hagler, A.T., J. Comput. Chem, **1994**, 15, 162, b) Famulari, A., Specchio, R., Sironi, M., Raimondi, M., *J. Chem. Phys.*, **1998**, 108, 3296, c) Mahoney, M. Jorgensen, W., *J. Chem. Phys.*, **2000**, 112, 8910.

23) Van Gunsteren, W. F., Mark, A. E., *J. Chem. Phys.*, **1998**, 108, 6109.

24) Gaussian 98, Revision A.7, Frisch, M. J., Trucks, G. W., Schlegel, H. B., Scuseria, G. E., Robb, M. A., Cheeseman, J. R., Zakrzewski, V. G., Montgomery, Jr., J. A.., Stratmann, R. E., Burant, J. C., Dapprich, S., Millam, J. M., Daniels, A. D., Kudin, K. N,. Strain, M. C, Farkas, O., Tomasi, J., Barone, V., Cossi, M., Cammi, R., Mennucci, B., Pomelli, C., Adamo, C., Clifford, S., Ochterski, J., Petersson, G. A., Ayala, P. Y., Cui, Q., Morokuma, K., Malick, D. K., Rabuck, A. D., Raghavachari, K., Foresman, J. B., Cioslowski, J., Ortiz, J. V., Baboul, A. G., Stefanov, B. B., Liu, G., Liashenko, A., Piskorz, P., Komaromi, I., Gomperts, R., Martin, R. L., Fox, D. J., Keith, T., Al-Laham, M. A., Peng, C. Y., Nanayakkara, A., Gonzalez, C., Challacombe, M., Gill, P. M. W., Johnson, B., Chen, W., Wong, M. W., Andres, J. L., Gonzalez, C., Head-Gordon, M., Replogle, E. S. Pople., J. A, Gaussian, Inc., Pittsburgh PA, 1998.

25) a) Becke, A. D., *J. Chem. Phys.*, **1993**, 98, 5648- 5652, b) Pople, J. A., Head-Gordon, M., Fox, D. J., Raghavachari, K. Curtiss, L. A., *J. Chem. Phys.*, **1989**, 90, 5622 c) Curtiss, L. A., Jones, C., Trucks, G. W., Raghavachari, K. Pople, J. A., *J. Chem. Phys.*, **1990**, 93, 2537.

26) Foresman, J. B. Frisch, Æ. *Exploring Chemistry with Electronic Structure Methods, Second Edition*, Gaussian Inc.: Pittsburgh, PA, 1996.

27) Klyne, W., Prelog, V., *Experientia*, **1960**, 16, 521.

28) Mayo, S. L., Olafson, B. D., Goddard III, W. A., *J. Phys. Chem.*, **1990**, 94, 8897.

29) Breneman, C. M., Wiberg, K. B., *J. Comp. Chem.*, **1990**, 11, 361.

30) a) Price, M. L. P., Ostrovsky, D., Jorgensen, W. L., *J. Comp. Chem.*, **2001**, 22, 1340, b) Rizzo, R. C., Jorgensen, W. L., *J. Am. Chem. Soc.*, **1999**, 121, 4827, c) Briggs, J. M., Nguyen, T. B.,. Jorgensen, W. L, *J. Phys. Chem.*, **1991**, 95, 3315, d) Jorgensen, W.L. et al., *J. Comput. Chem.*, **1993**, 14, 206.

31) a) Siepmann J.I., Karaborni S., Smit B., *Nature*, **1993**, 365, 330 b) Jorgensen W.L., Madura J.D., Swenson C.J., *J. Am. Chem. Soc.*, **1994**, 106, 6638.

32) a) Hourihan, M. O'Reilly Network: The Sanctity of Elements, or Why You Shouldn't be Double-clicking in a <textarea> [May 3, 2002]. http://www.oreillynet.com/pub/a/javascript/2002/05/03/megnut.html (accessed May, 2002), b) Rosenfeld, L., Morville, P., *Information Architecture for the World Wide Web: Designing Large-scale Web Sites*, O'Reilly & Associates: Sebastopol, California, 1998, c) Rational Software. http://www.rational.com (accessed May 2002), d) R. S. Pressman and Associates, Inc. RSP&A Project Planning and Management. http://www.rspa.com/spi/project-mgmt.html (accessed May 2002).

33) CPAN. CPAN/ports. www.perl.com/CPAN/ports/ (accessed May 2002).

34) Cozens, S. perl.com: Ten Perl Myths [Feb. 23, 2000]. http://www.perl.com/pub/a/2000/01/10PerlMyths.html (accessed May 2002),

35) See, for example, some of the programs from reference 6.

36) 1) Garciá, E., MATCHEM: A symbolic model for the computer representation and manipulation of chemical structures and reactions, PhD Thesis, Instituto de Qui'micha, Universidade de Brasilia, Brasilia DF, Brasil, 1994, b) Garciá, E., Instituto de Qui'micha, Universidade de Brasilia, Brasilia DF, Brasil, personal communications, c) Garciá, E., submitted to *J. Chem. Inf. and Comp. Sci.*, **2002**.

37) a) Perldoc.com. perlXStut. http://www.perldoc.com/perl5.6.1/pod/perlxstut.html (accessed May 2002), b) Perldoc.com. perlxs. http://www.perldoc.com/perl5.6.1/pod/perlxs.html (accessed May 2002).

38) a) Cahn, R.S., Ingold, C.K. Prelog, V., *Angew. Chem.*, **1966**, 78, 413, b) Cahn, R.S., Ingold, C.K. Prelog, V., *Angew. Chem. Internat. Ed. Eng.*, **1966**, 5, 385; c) Prelog, V., Helmchen, G., *Angew. Chem.*, **1982**, 94, 614-631, d) Prelog, V., Helmchen, G., *Angew. Chem. Internat. Ed. Eng.* , **1982**, 21, 567.

39) Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language User Guide*, Addison-Wesley: Reading, Massachusetts, 1999.

40) Wall, L., Christiansen, T., Orwant, J., *Programming Perl, 3rd Edition*, O'Reilly & Associates: Sebastopol, California, 2000.

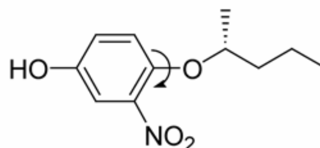
41) The Free Software Foundation. Licenses – GNU Project – Free Software Foundation (FSF). http://www.gnu.org/licenses/licenses.html (accessed May 2002).

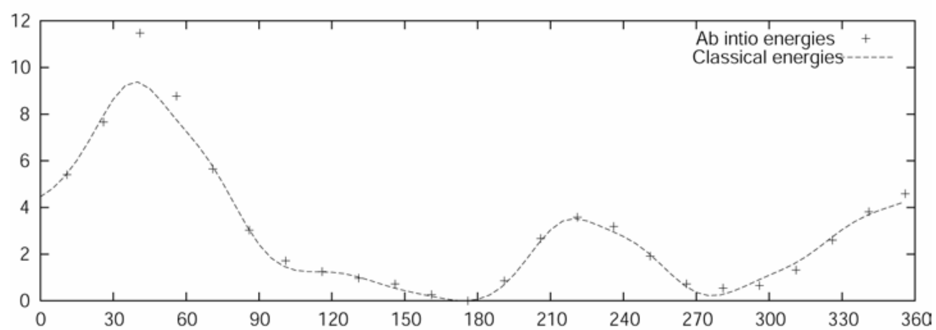
Appendix A

This appendix contains the actual fitting data for all of the dihedrals that went into the final force field, as well as the actual parameters of the cosine series we use to reproduce the ab initio torsional profile.

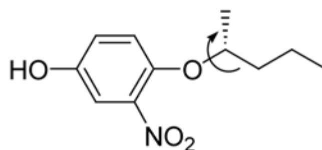
Compound 1

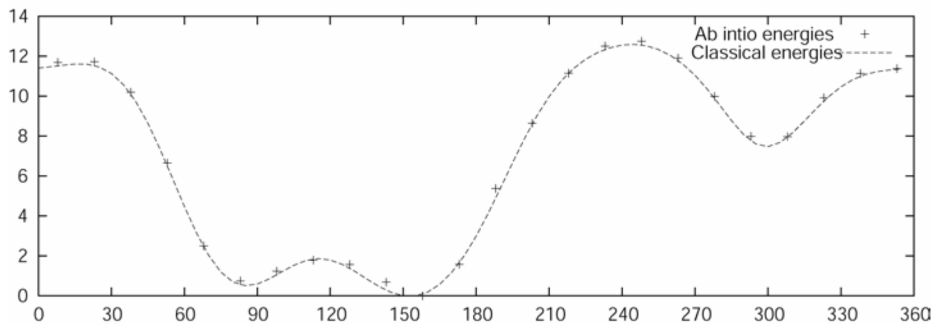
C11H15NO4-0_5-6-15-16
-8.196 15.993 -21.255 -31.435 129.370 26.806 -96.932
Classical energy offset: 0.0852031



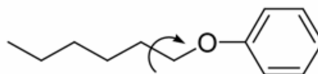


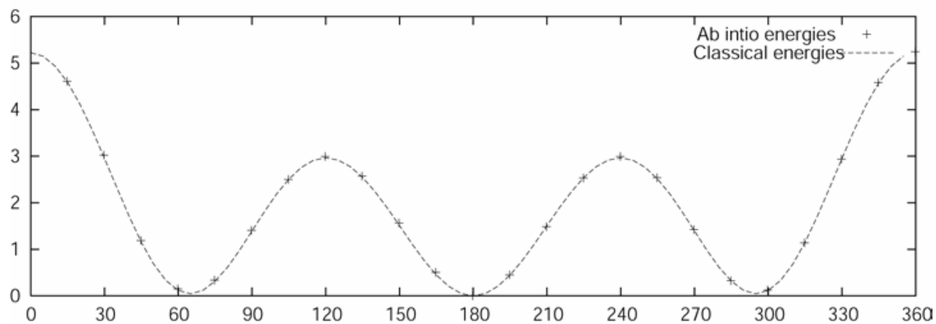
C11H15NO4-0_6-15-16-17
-2.521 -2.112 27.228 13.334 -47.415 -9.295 21.565
Classical energy offset: 0.130546



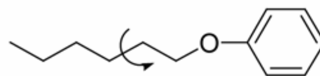


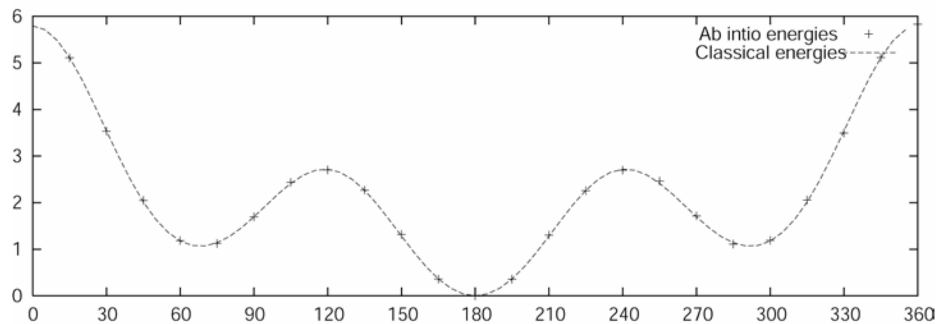
C12H18O-1_12-13-14-15
-0.429 -4.752 0.170 7.701 1.889 -0.372 -0.875
Classical energy offset: 0.0174484



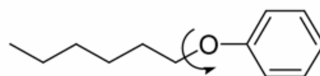


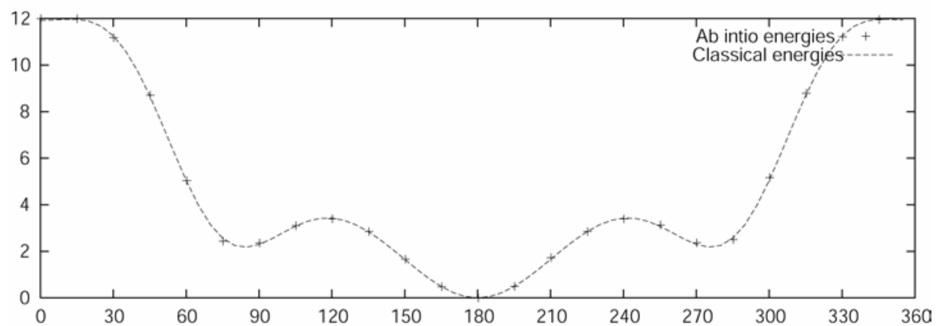
C12H18O-1_13-14-15-16
-0.138 -2.789 1.310 4.761 -1.586 0.907 1.459
Classical energy offset: 0.00544838



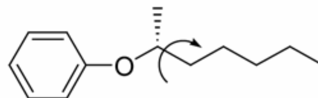


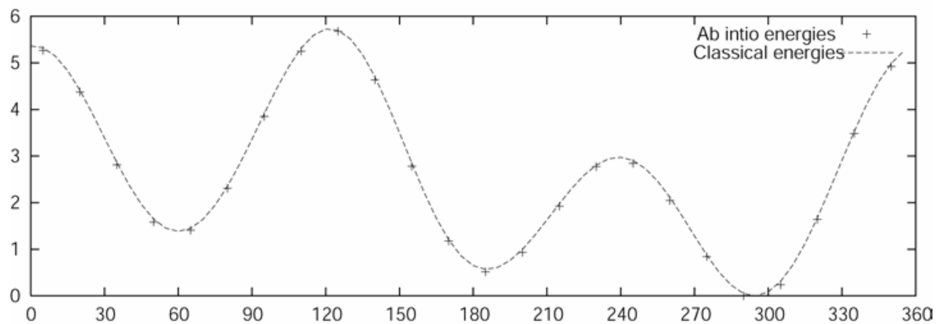
C12H18O-1_4-12-13-14
0.068 -4.233 2.736 10.958 -1.124 -6.567 -3.351
Classical energy offset: 0.0104484



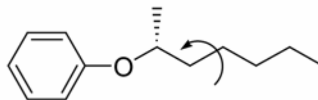


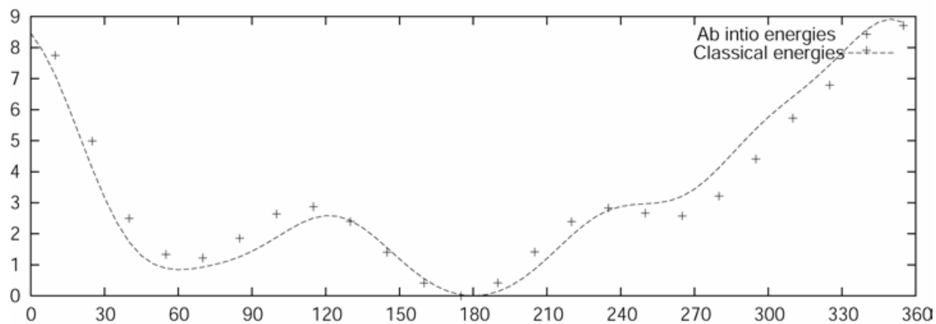
C13H20O-0_12-13-14-15
0.309 3.525 -18.230 -3.003 54.961 1.140 -36.870
Classical energy offset: -0.0511571



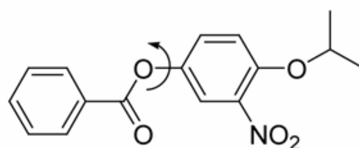


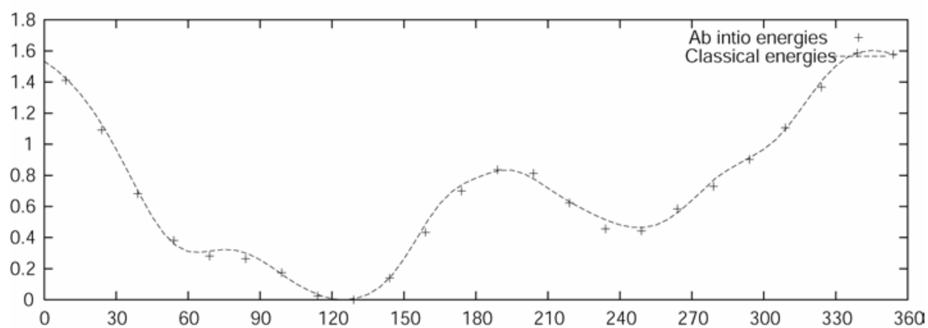
C13H20O-0_13-14-15-16
1.023 -2.371 2.089 0.376 -7.437 3.638 5.574
Classical energy offset: 0.129162



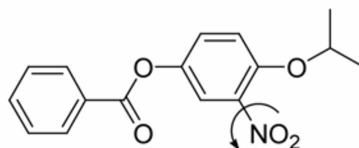


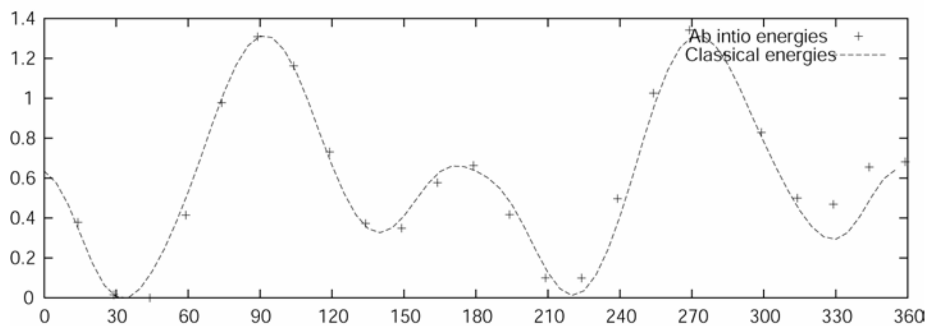
C16H15NO5-0_12-14-15-16
62.395 19.721 -9.194 20.015 8.417 -10.874 -2.108
Classical energy offset: -0.0139798



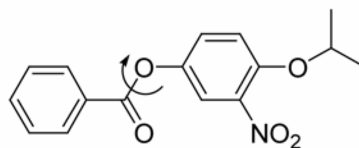


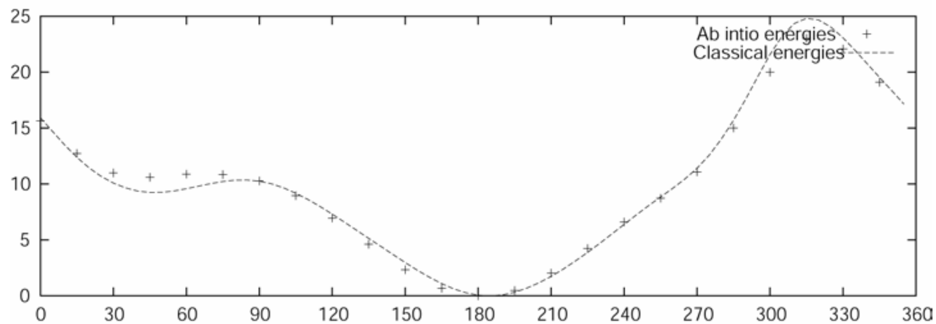
C16H15NO5-0_16-17-24-25
-97.394 -0.000 89.828 0.000 20.218 -0.000 -16.240
Classical energy offset: 0.0200807



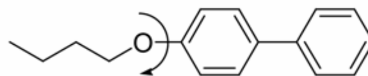


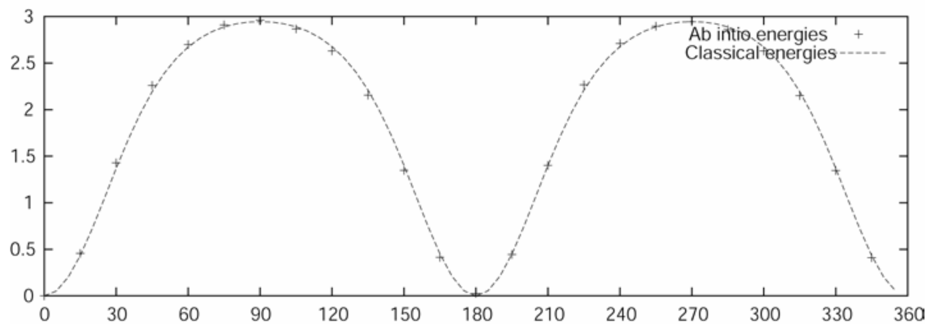
C16H15NO5-0_5-12-14-15
71.287 0.722 -14.007 -1.425 7.286 2.641 -2.044
Classical energy offset: -0.117573



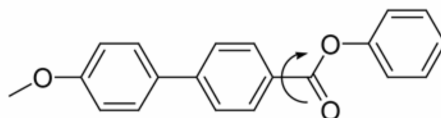


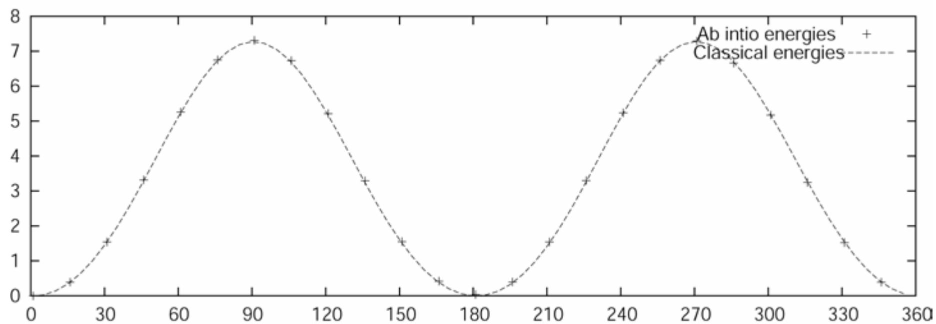
C16H18O-0_3-4-22-23
2.935 0.000 -1.282 -0.000 -2.115 0.000 -1.380
Classical energy offset: -0.00255197



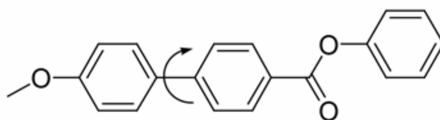


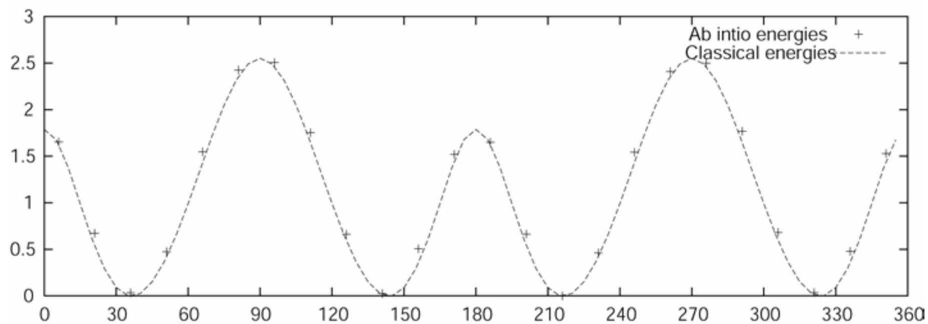
C20H16O3-0_13-14-21-22
43.527 -0.000 -75.258 0.000 -8.680 -0.000 5.587
Classical energy offset: 0.0173037



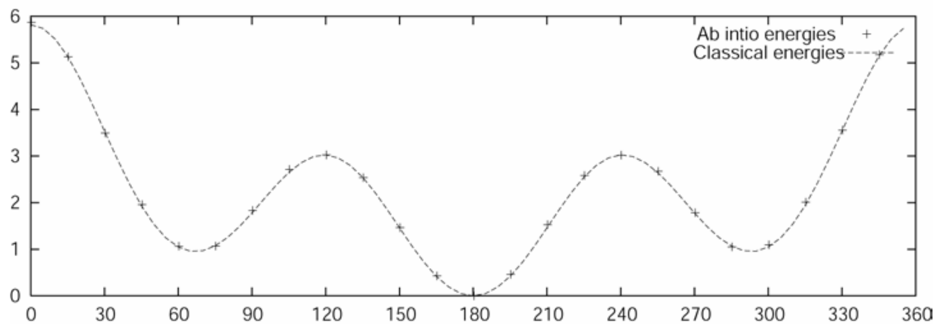


C20H16O3-0_2-1-11-12
-2.526 0.675 -8.070 -2.021 6.211 1.311 -3.533
Classical energy offset: 0.0724218

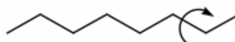


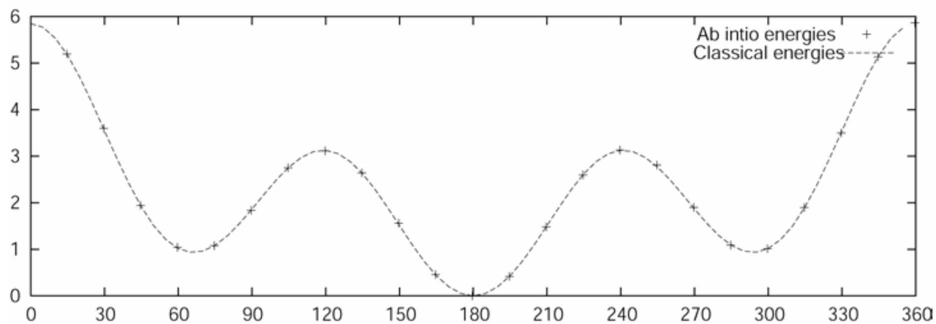


C7H16O-1_4-5-6-7
1.800 -3.434 1.272 5.663 -1.270 0.659 1.106
Classical energy offset: 0.02

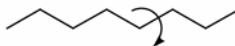


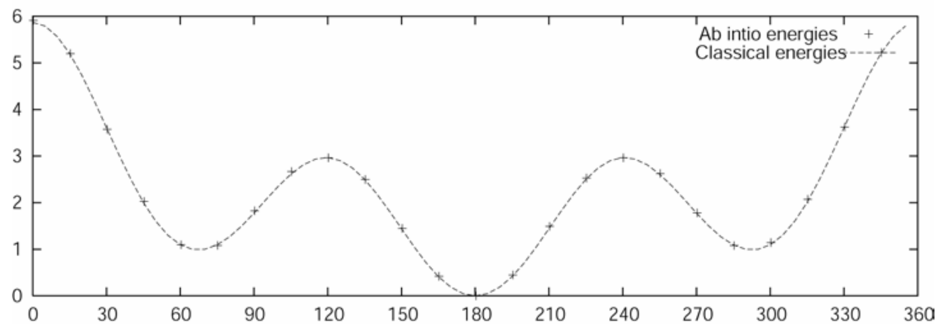
C8H18-0_1-2-3-4
1.871 -3.646 0.937 6.003 -0.667 0.554 0.775
Classical energy offset: 0.005



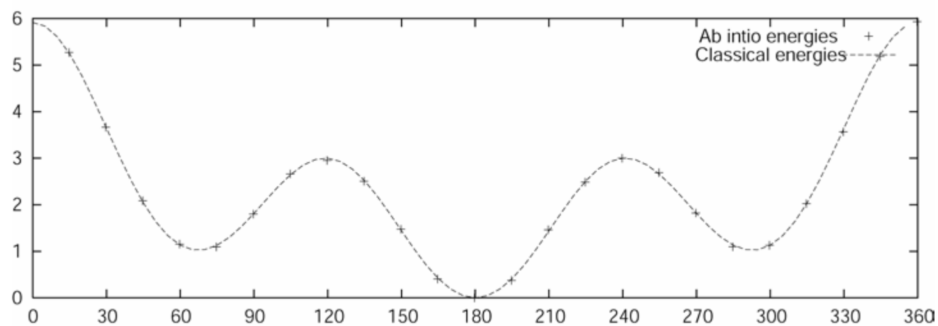


C8H18-0_2-3-4-5
1.792 -3.311 1.202 5.595 -0.966 0.629 0.900
Classical energy offset: 0.015

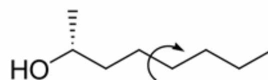


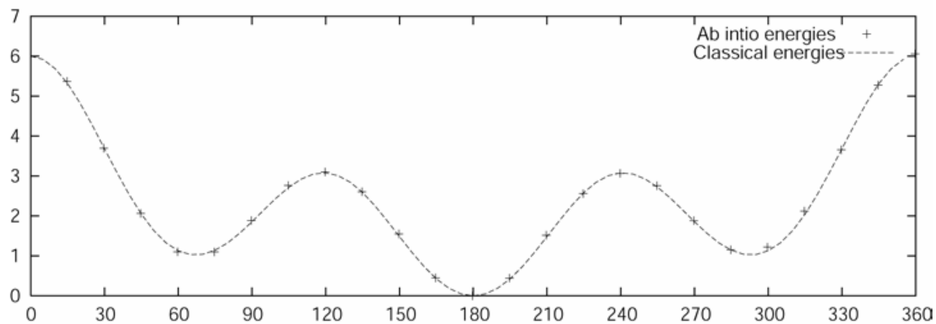


C8H18-0_3-4-5-6
1.809 -3.322 1.225 5.656 -1.127 0.602 1.024
Classical energy offset: -0.005

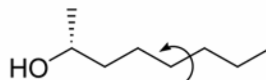


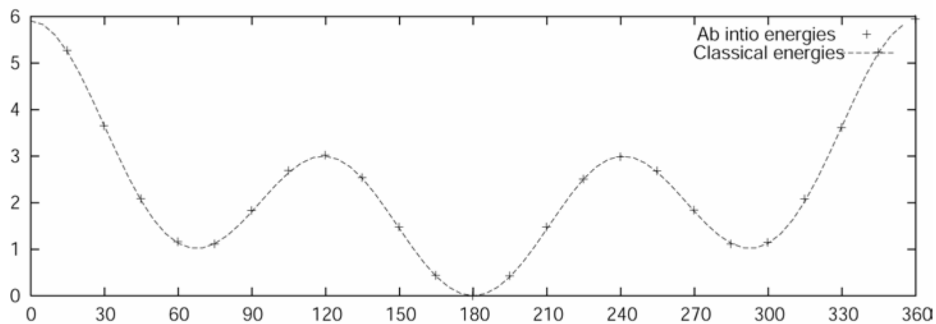
C8H18O-0_4-5-6-7
1.714 -3.440 1.190 5.795 -1.045 0.628 0.982
Classical energy offset: 0.0188579





C8H18O-0_5-6-7-8
1.669 -3.330 1.219 5.684 -1.085 0.578 0.986
Classical energy offset: 0.0178579

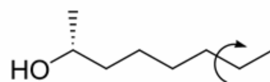


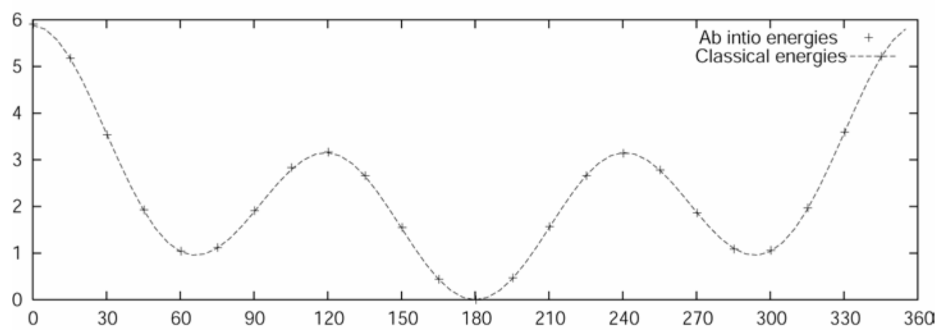


C8H18O-0_6-7-8-10

1.730 -3.636 0.985 5.902 -0.716 0.665 0.781

Classical energy offset: 0.00985791





Appendix B

This appendix includes a graphical summary of the fragmentation of all three compounds, as well as a graphical summary of the atom and bond mapping for compound 1.

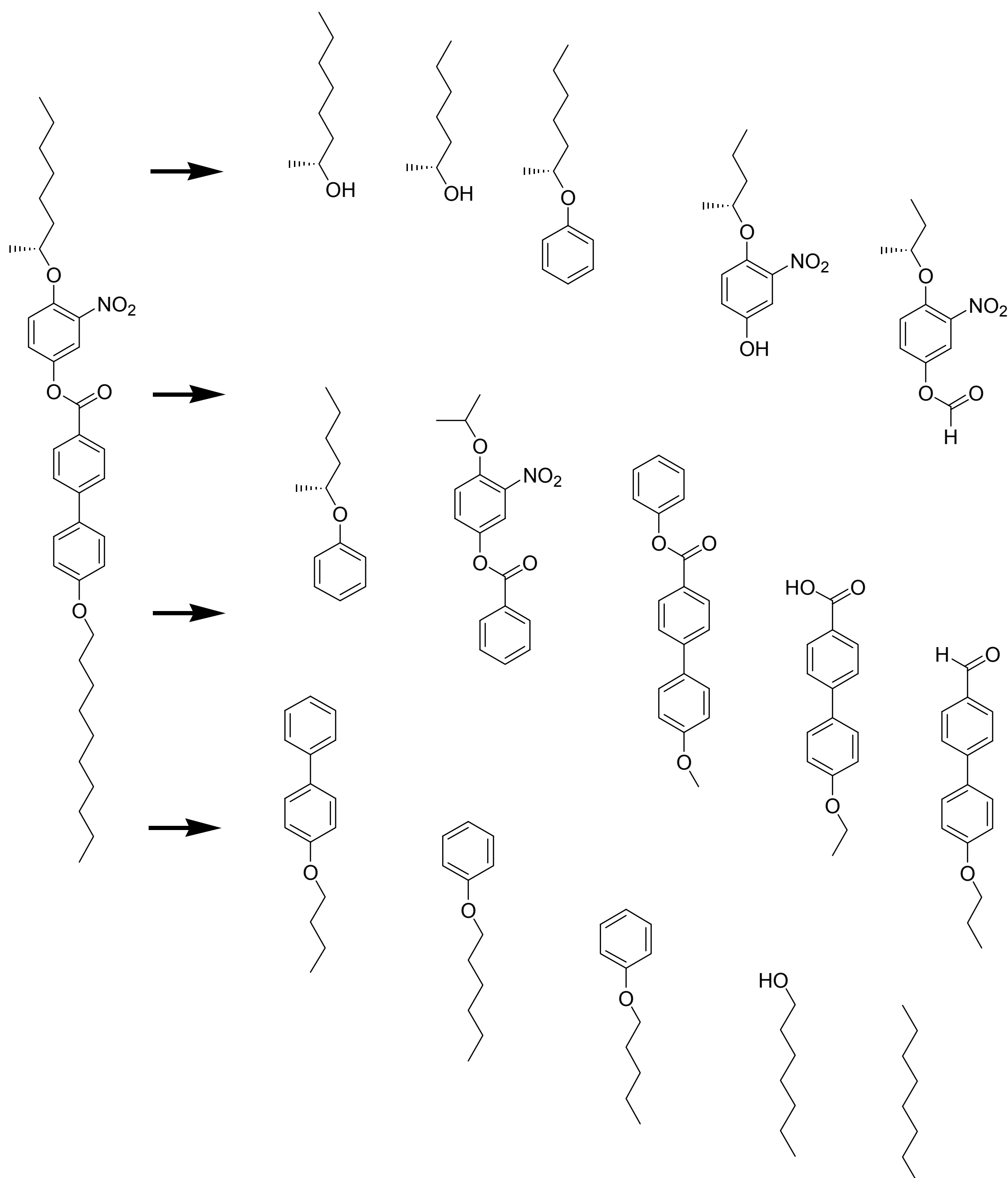
Note that in all of the mappings, the standard skeletal structure is presented, with omitted hydrogens in almost all cases. The actual atom and bond maps contained descriptions for mapping between all atoms and bonds.

In the Parent to Fragment Atom Mappings, all hydrogens mapped with the carbon they were attached to. Additionally, some selections are with a box (when there are multiple atoms that go to a single fragment), and some selections are directly from a parent atom to a fragment atom.

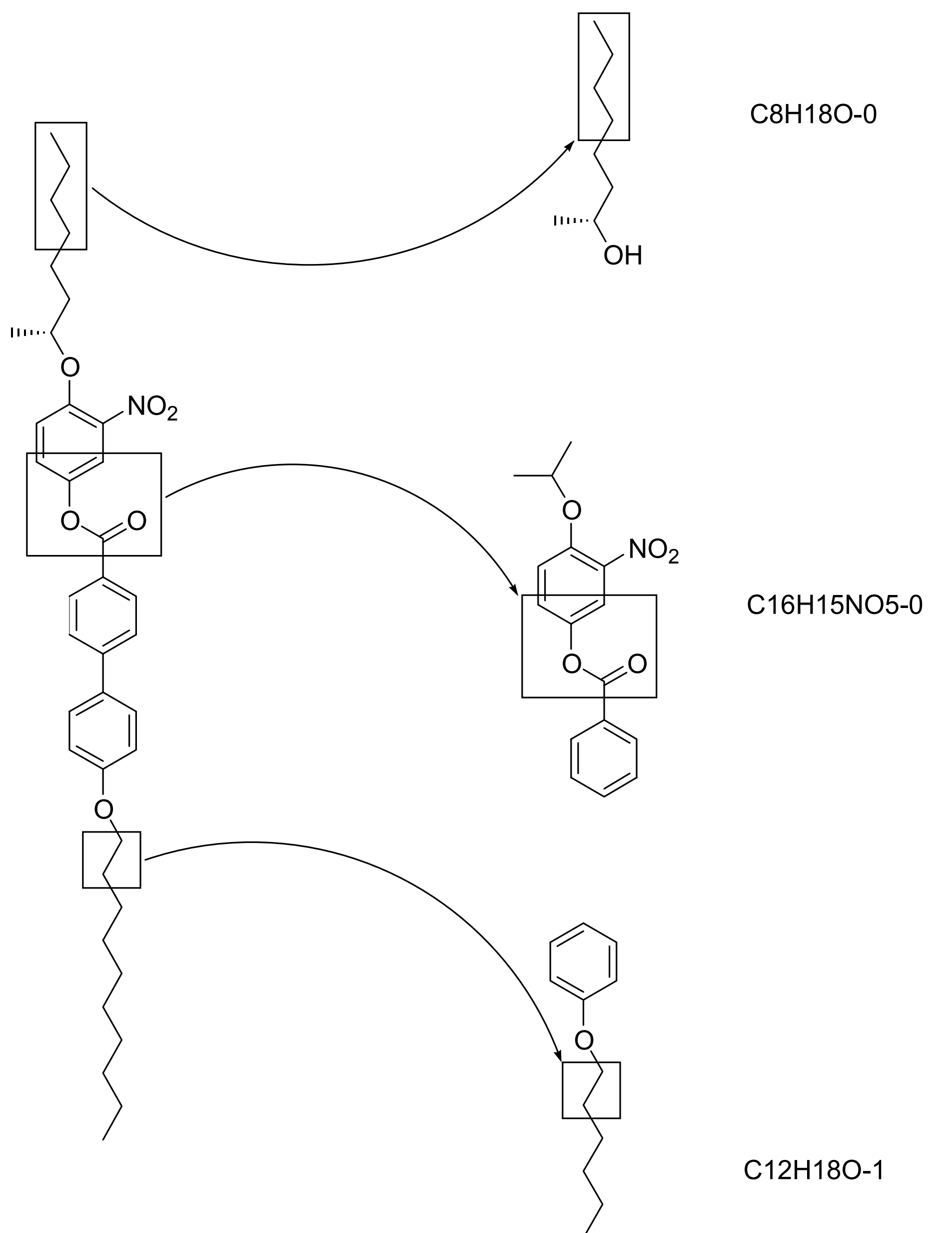
In the Parent to Fragment Bond Mappings, only bonds between heavy atoms are illustrated, for clarity. Also, they are matched by letter, instead of using arrows to indicate the correlation.

After the initial graphical presentation, the relevant sections of the atom and bond mapping sections of the output from qdb_check are included, as an example. The fragments will be in the sample database included on the CD. Remember, the atoms and bond numbering starts from 0, instead of 1, so depending on what program you use to visualize the molecules, you may need to add one to the values in the atom and bond maps.

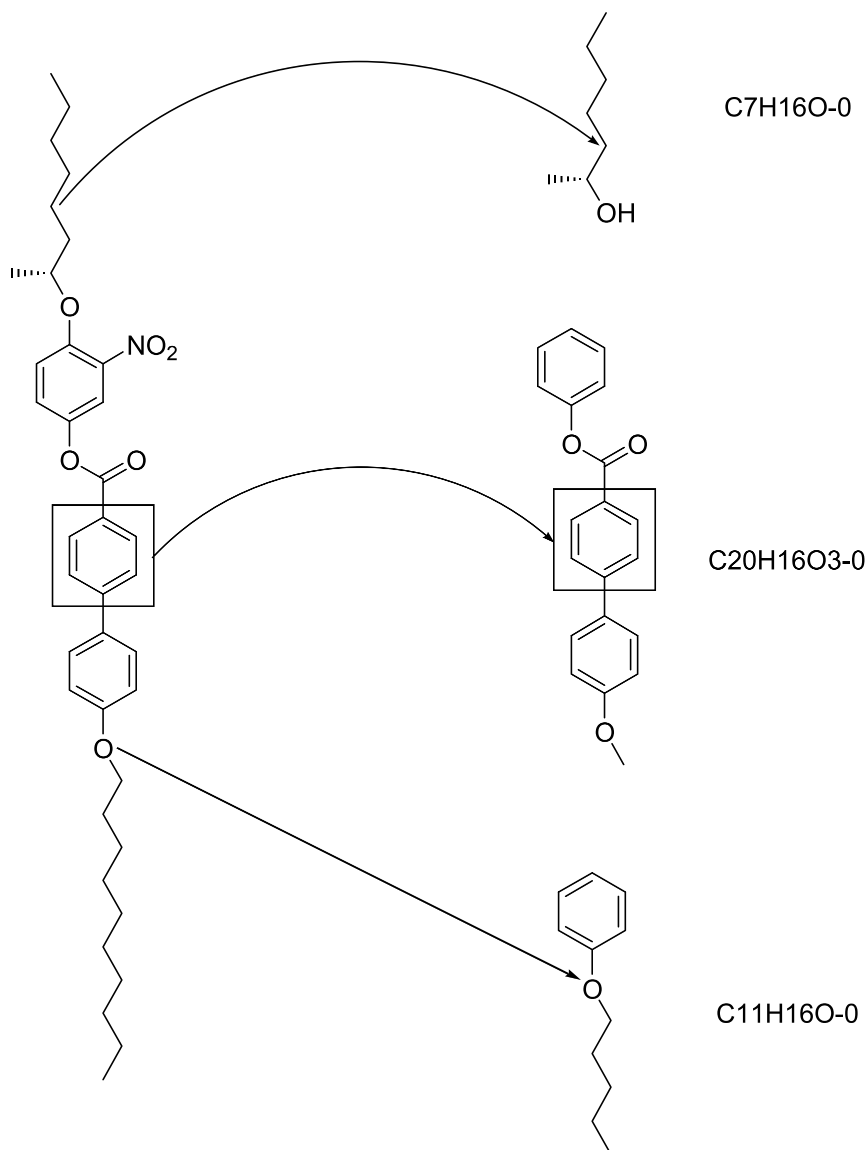
Fragmentation of Compound **1**



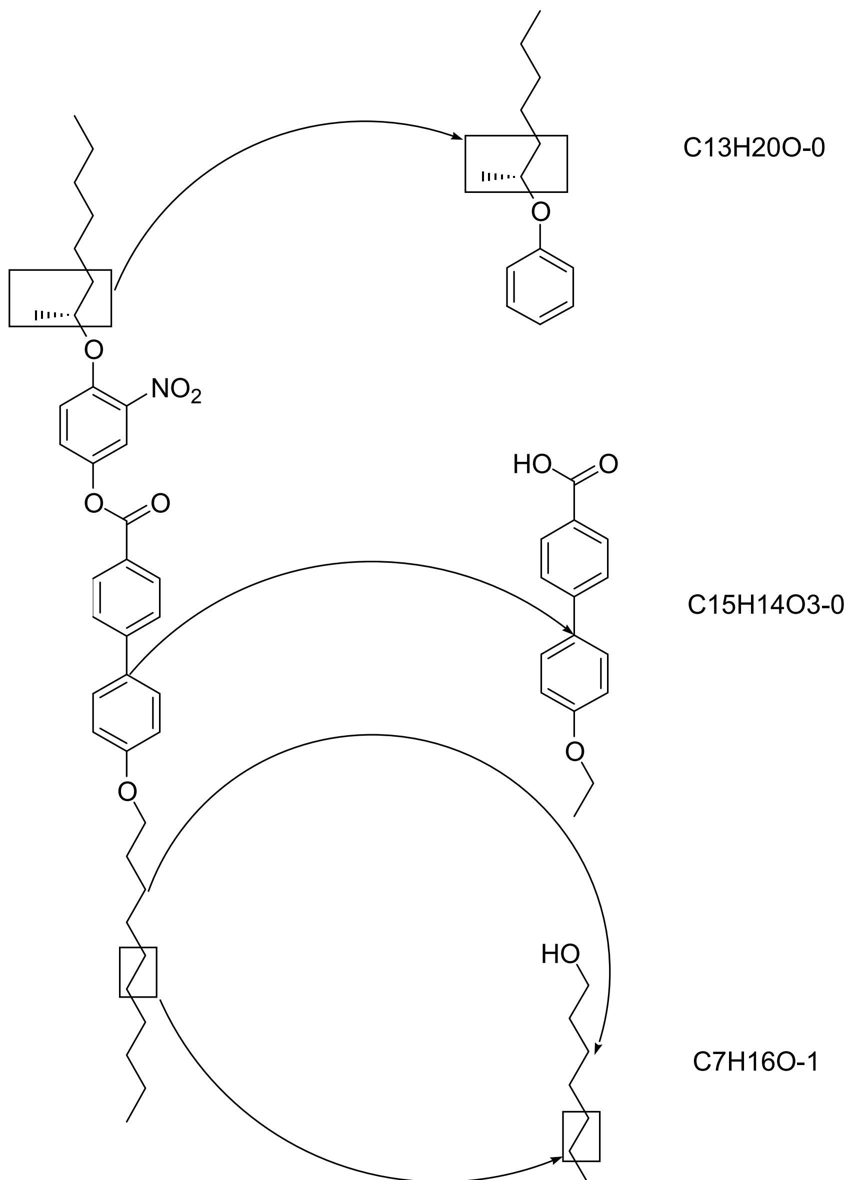
Parent to Fragment Atom Mappings for Parent compound 1, Part 1



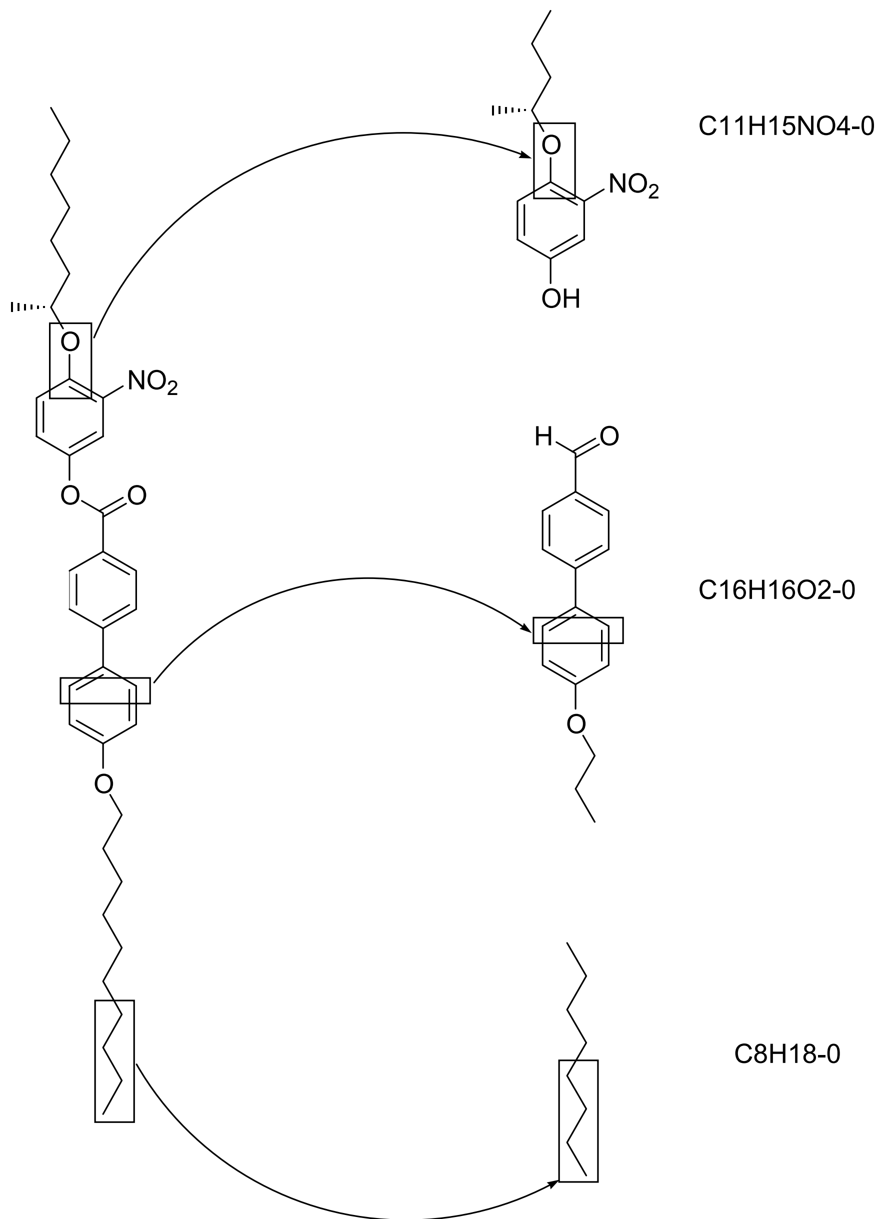
Parent to Fragment Atom Mappings for Parent compound 1, Part 2



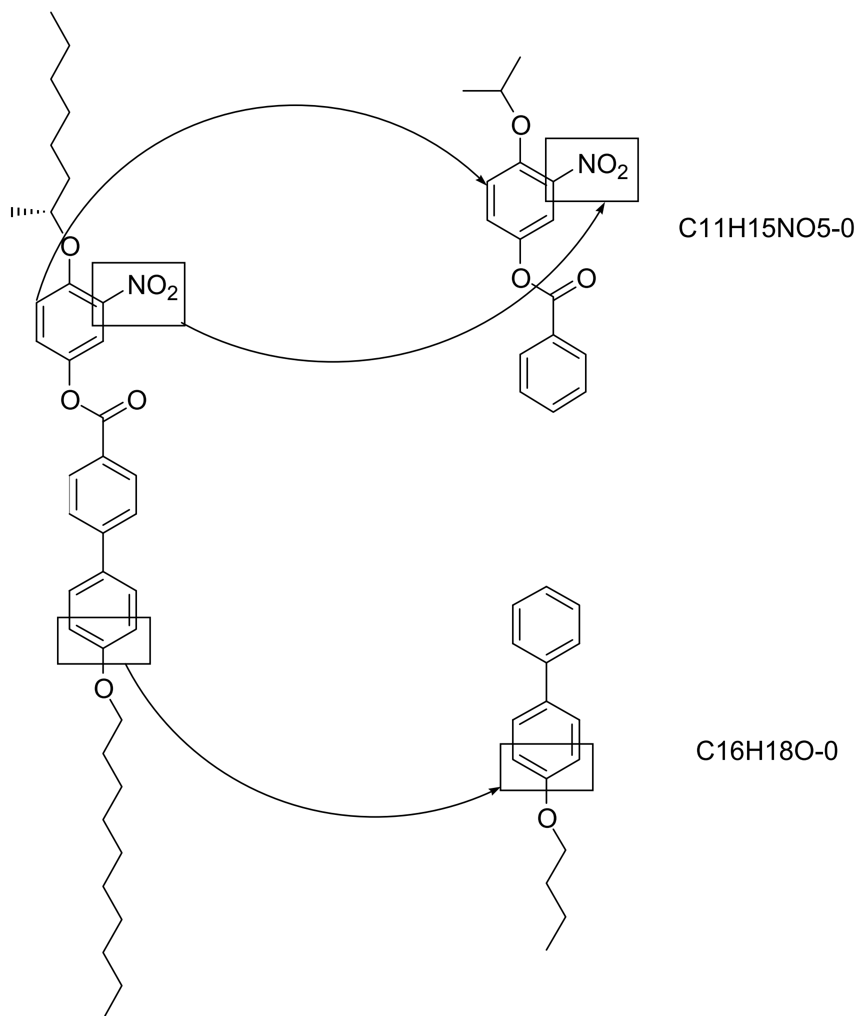
Parent to Fragment Atom Mappings for Parent compound 1, Part 3



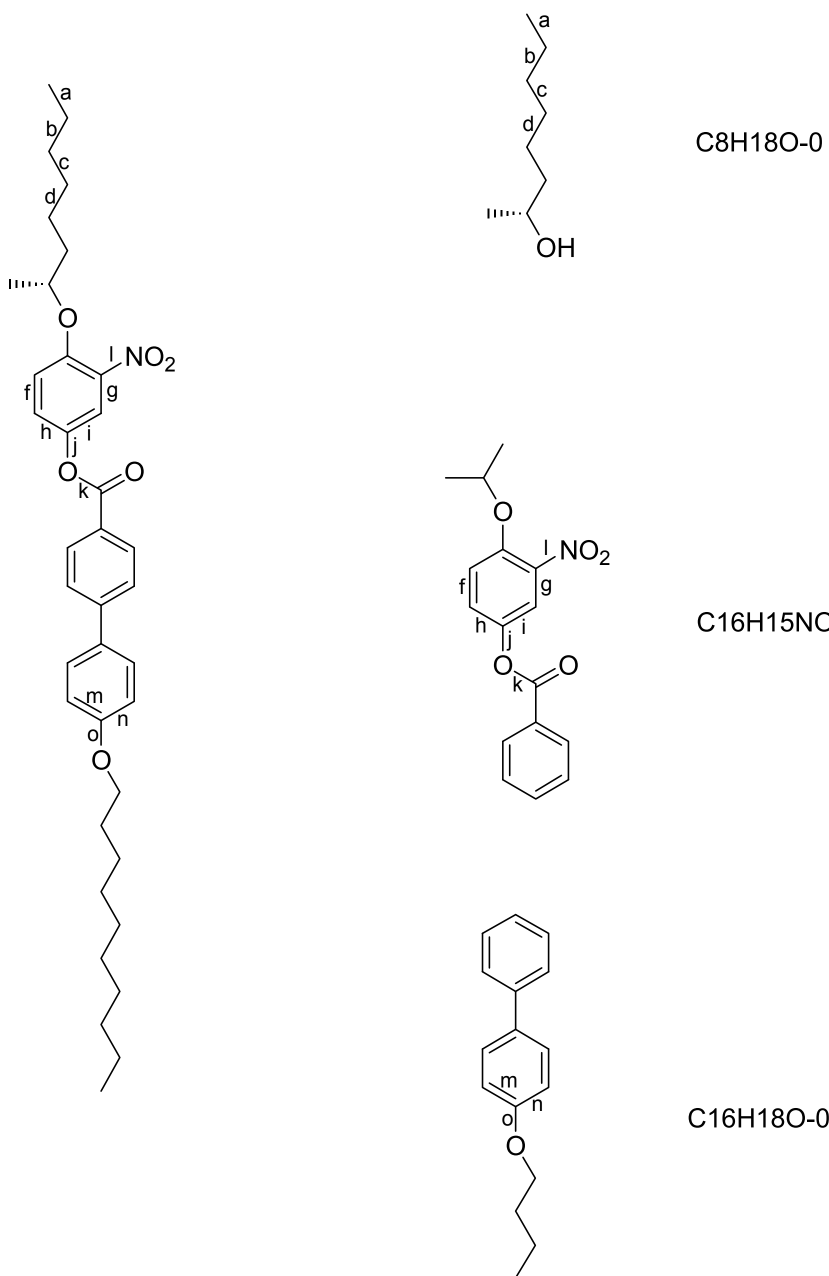
Parent to Fragment Atom Mappings for Parent compound 1, Part 4



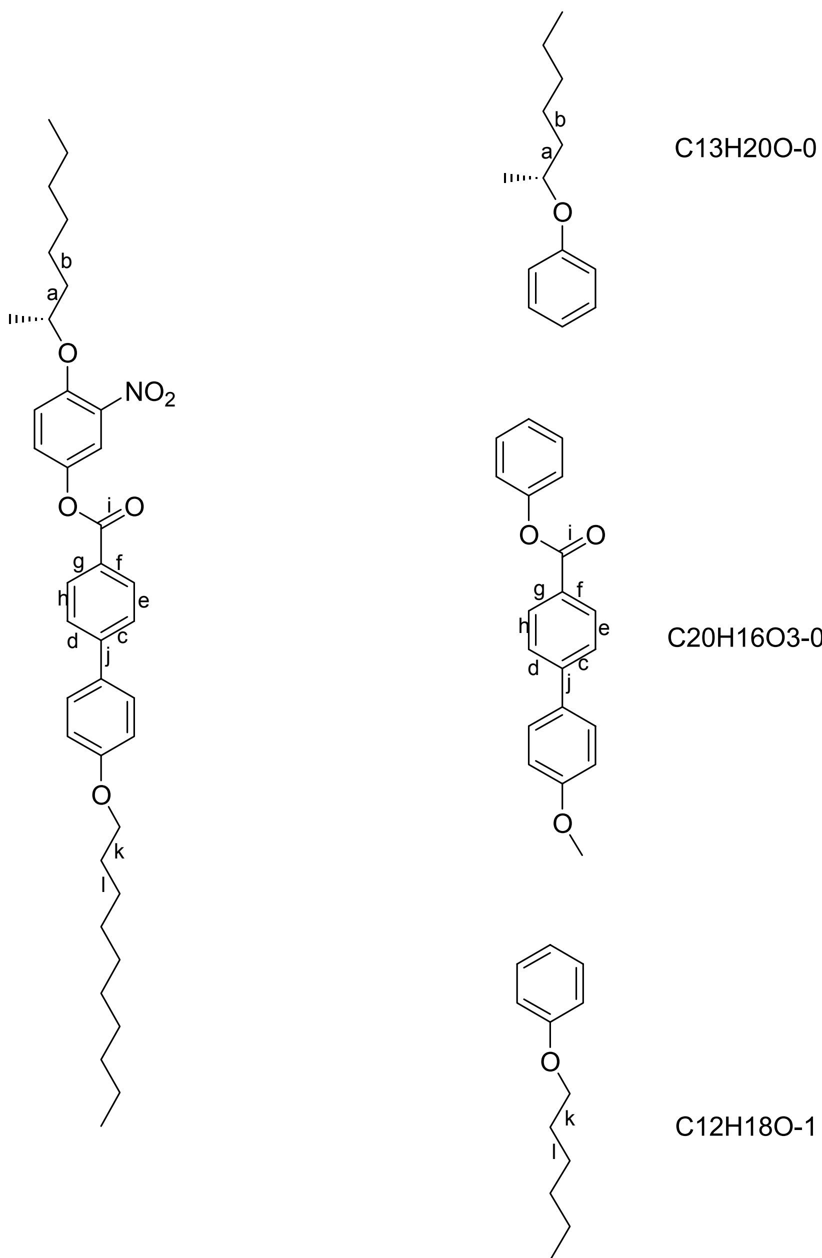
Parent to Fragment Atom Mappings for Parent compound 1, Part 5



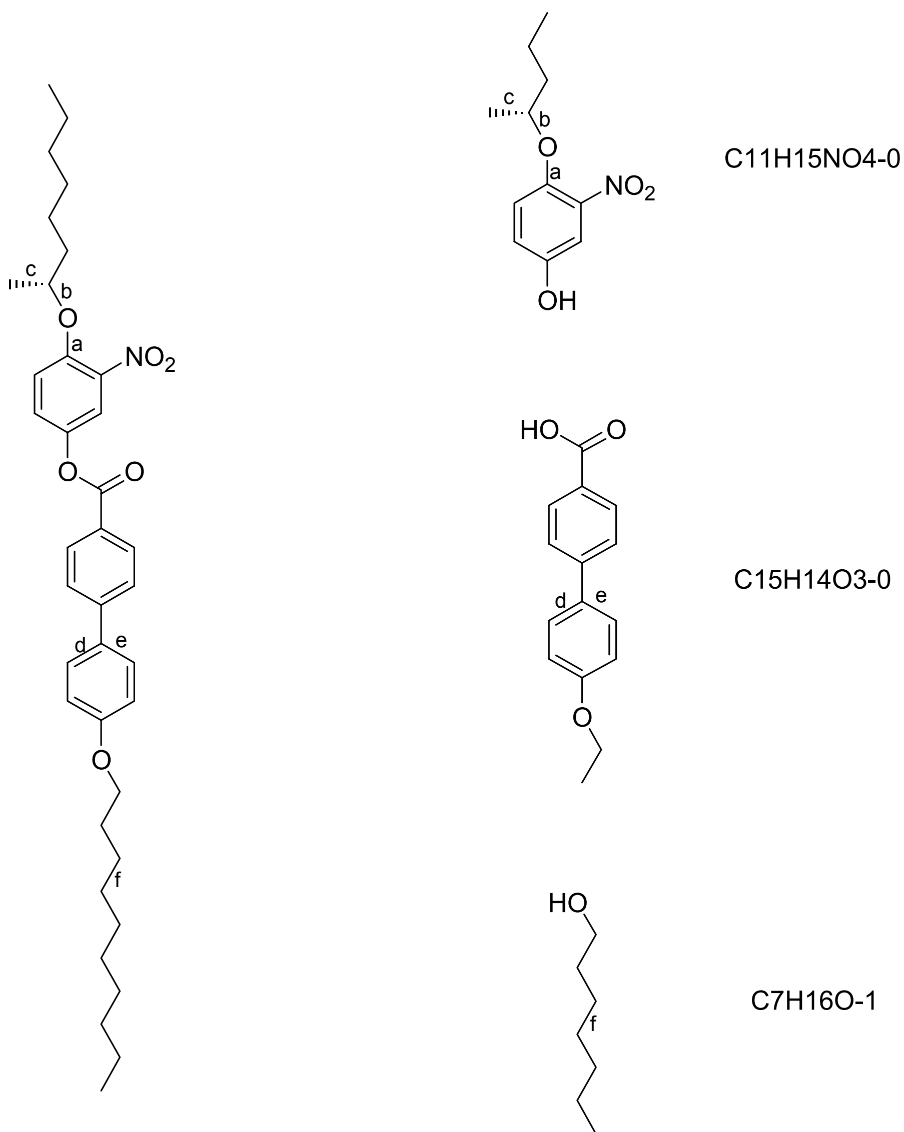
Parent to Fragment Bond Mappings for Parent compound 1, Part 1



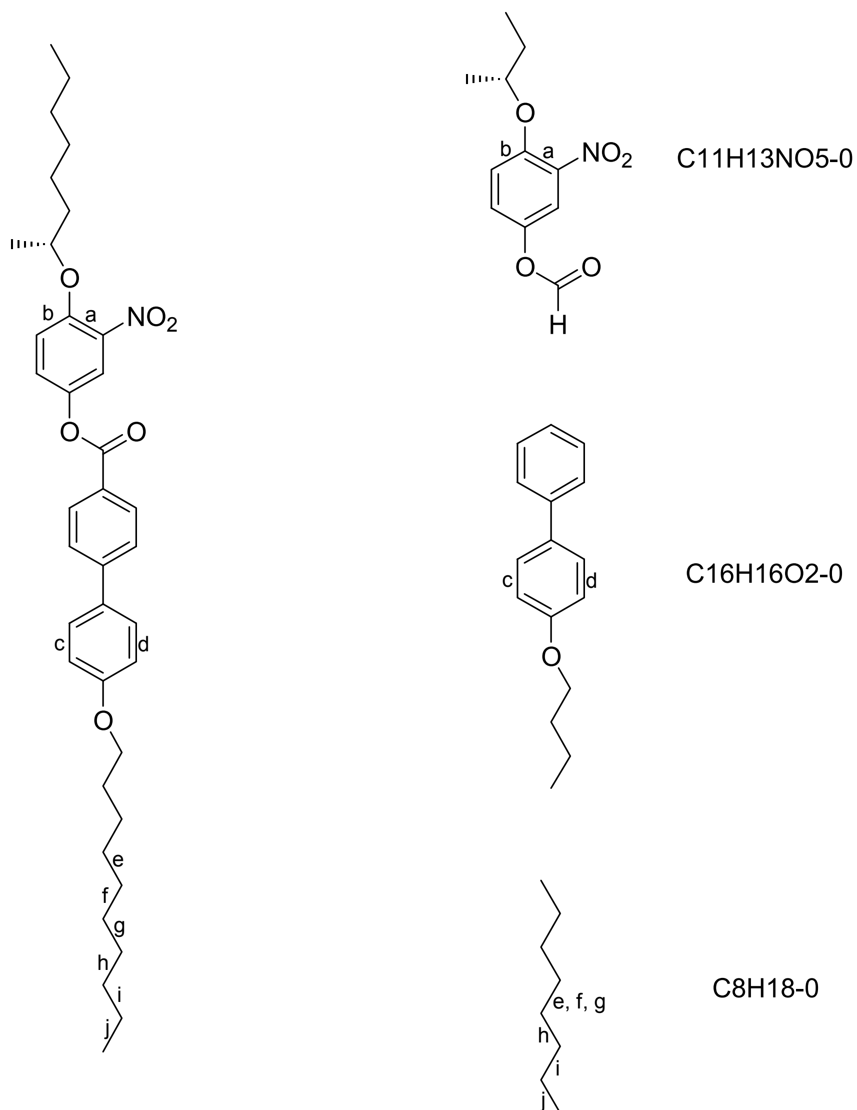
Parent to Fragment Bond Mappings for Parent compound 1, Part 2



Parent to Fragment Bond Mappings for Parent compound 1, Part 3



Parent to Fragment Bond Mappings for Parent compound 1, Part 4



Atom Map List for Compound 1

```
Dir: C15H14O3-0 Parent atom 0: Qdb atom 0: qdb
Dir: C16H16O2-0 Parent atom 1: Qdb atom 1: qdb
Dir: C16H18O-0 Parent atom 2: Qdb atom 2: qdb
Dir: C16H18O-0 Parent atom 3: Qdb atom 3: qdb
Dir: C16H18O-0 Parent atom 4: Qdb atom 2: qdb
Dir: C16H16O2-0 Parent atom 5: Qdb atom 1: qdb
Dir: C16H16O2-0 Parent atom 6: Qdb atom 9: qdb
Dir: C16H18O-0 Parent atom 7: Qdb atom 7: qdb
Dir: C16H18O-0 Parent atom 8: Qdb atom 7: qdb
Dir: C16H16O2-0 Parent atom 9: Qdb atom 9: qdb
Dir: C20H16O3-0 Parent atom 10: Qdb atom 10: qdb
Dir: C20H16O3-0 Parent atom 11: Qdb atom 15: qdb
Dir: C20H16O3-0 Parent atom 12: Qdb atom 12: qdb
Dir: C20H16O3-0 Parent atom 13: Qdb atom 13: qdb
Dir: C20H16O3-0 Parent atom 14: Qdb atom 12: qdb
Dir: C20H16O3-0 Parent atom 15: Qdb atom 15: qdb
Dir: C20H16O3-0 Parent atom 16: Qdb atom 16: qdb
Dir: C20H16O3-0 Parent atom 17: Qdb atom 17: qdb
Dir: C20H16O3-0 Parent atom 18: Qdb atom 17: qdb
Dir: C20H16O3-0 Parent atom 19: Qdb atom 16: qdb
Dir: C16H15NO5-0 Parent atom 20: Qdb atom 11: qdb
Dir: C16H15NO5-0 Parent atom 21: Qdb atom 12: qdb
Dir: C16H15NO5-0 Parent atom 22: Qdb atom 13: qdb
Dir: C16H15NO5-0 Parent atom 23: Qdb atom 14: qdb
Dir: C16H15NO5-0 Parent atom 24: Qdb atom 15: qdb
Dir: C11H13NO5-0 Parent atom 25: Qdb atom 6: qdb
Dir: C11H15NO4-0 Parent atom 26: Qdb atom 5: qdb
Dir: C11H13NO5-0 Parent atom 27: Qdb atom 8: qdb
Dir: C16H15NO5-0 Parent atom 28: Qdb atom 19: qdb
Dir: C16H15NO5-0 Parent atom 29: Qdb atom 20: qdb
Dir: C11H13NO5-0 Parent atom 30: Qdb atom 11: qdb
Dir: C16H15NO5-0 Parent atom 31: Qdb atom 22: qdb
Dir: C11H13NO5-0 Parent atom 32: Qdb atom 13: qdb
Dir: C11H13NO5-0 Parent atom 33: Qdb atom 14: qdb
Dir: C11H13NO5-0 Parent atom 34: Qdb atom 14: qdb
Dir: C11H16O-0 Parent atom 35: Qdb atom 11: qdb
Dir: C11H15NO4-0 Parent atom 36: Qdb atom 14: qdb
Dir: C13H20O-0 Parent atom 37: Qdb atom 12: qdb
Dir: C13H20O-0 Parent atom 38: Qdb atom 13: qdb
Dir: C13H20O-0 Parent atom 39: Qdb atom 14: qdb
Dir: C7H16O-0 Parent atom 40: Qdb atom 5: qdb
Dir: C13H20O-0 Parent atom 41: Qdb atom 16: qdb
Dir: C13H20O-0 Parent atom 42: Qdb atom 16: qdb
Dir: C8H18O-0 Parent atom 43: Qdb atom 8: qdb
Dir: C7H16O-0 Parent atom 44: Qdb atom 10: qdb
Dir: C7H16O-0 Parent atom 45: Qdb atom 10: qdb
Dir: C8H18O-0 Parent atom 46: Qdb atom 11: qdb
Dir: C8H18O-0 Parent atom 47: Qdb atom 11: qdb
Dir: C8H18O-0 Parent atom 48: Qdb atom 13: qdb
Dir: C8H18O-0 Parent atom 49: Qdb atom 14: qdb
Dir: C8H18O-0 Parent atom 50: Qdb atom 14: qdb
Dir: C8H18O-0 Parent atom 51: Qdb atom 16: qdb
Dir: C8H18O-0 Parent atom 52: Qdb atom 17: qdb
```

```
Dir: C8H18O-0 Parent atom 53: Qdb atom 17: qdb
Dir: C12H18O-1 Parent atom 54: Qdb atom 12: qdb
Dir: C12H18O-1 Parent atom 55: Qdb atom 13: qdb
Dir: C12H18O-1 Parent atom 56: Qdb atom 13: qdb
Dir: C12H18O-1 Parent atom 57: Qdb atom 15: qdb
Dir: C12H18O-1 Parent atom 58: Qdb atom 16: qdb
Dir: C12H18O-1 Parent atom 59: Qdb atom 16: qdb
Dir: C7H16O-1 Parent atom 60: Qdb atom 8: qdb
Dir: C7H16O-1 Parent atom 61: Qdb atom 10: qdb
Dir: C7H16O-1 Parent atom 62: Qdb atom 10: qdb
Dir: C8H18O-0 Parent atom 63: Qdb atom 8: qdb
Dir: C8H18O-0 Parent atom 64: Qdb atom 11: qdb
Dir: C8H18O-0 Parent atom 65: Qdb atom 11: qdb
Dir: C7H16O-1 Parent atom 66: Qdb atom 11: qdb
Dir: C7H16O-1 Parent atom 67: Qdb atom 12: qdb
Dir: C7H16O-1 Parent atom 68: Qdb atom 12: qdb
Dir: C7H16O-1 Parent atom 69: Qdb atom 11: qdb
Dir: C7H16O-1 Parent atom 70: Qdb atom 12: qdb
Dir: C7H16O-1 Parent atom 71: Qdb atom 12: qdb
Dir: C8H18-0 Parent atom 72: Qdb atom 10: qdb
Dir: C8H18-0 Parent atom 73: Qdb atom 11: qdb
Dir: C8H18-0 Parent atom 74: Qdb atom 11: qdb
Dir: C8H18-0 Parent atom 75: Qdb atom 16: qdb
Dir: C8H18-0 Parent atom 76: Qdb atom 17: qdb
Dir: C8H18-0 Parent atom 77: Qdb atom 17: qdb
Dir: C8H18-0 Parent atom 78: Qdb atom 4: qdb
Dir: C8H18-0 Parent atom 79: Qdb atom 20: qdb
Dir: C8H18-0 Parent atom 80: Qdb atom 20: qdb
Dir: C13H20O-0 Parent atom 81: Qdb atom 29: qdb
Dir: C13H20O-0 Parent atom 82: Qdb atom 30: qdb
Dir: C13H20O-0 Parent atom 83: Qdb atom 30: qdb
Dir: C13H20O-0 Parent atom 84: Qdb atom 30: qdb
Dir: C8H18O-0 Parent atom 85: Qdb atom 23: qdb
Dir: C8H18O-0 Parent atom 86: Qdb atom 26: qdb
Dir: C8H18O-0 Parent atom 87: Qdb atom 26: qdb
Dir: C8H18O-0 Parent atom 88: Qdb atom 26: qdb
Dir: C8H18-0 Parent atom 89: Qdb atom 22: qdb
Dir: C8H18-0 Parent atom 90: Qdb atom 0: qdb
Dir: C8H18-0 Parent atom 91: Qdb atom 0: qdb
Dir: C8H18-0 Parent atom 92: Qdb atom 0: qdb
```

Bond Map List for Compound 1

```
Dir: C15H14O3-0 Parent bond 0-1: Qdb bond 0-1: qdb homo -
Dir: C15H14O3-0 Parent bond 0-5: Qdb bond 0-1: qdb homo
Dir: C20H16O3-0 Parent bond 0-10: Qdb bond 0-10: qdb homo -
Dir: C16H16O2-0 Parent bond 1-2: Qdb bond 4-5: qdb homo -
Dir: C16H16O2-0 Parent bond 1-6: Qdb bond 5-9: qdb homo +
Dir: C16H18O-0 Parent bond 2-3: Qdb bond 2-3: qdb homo -
Dir: C16H18O-0 Parent bond 2-7: Qdb bond 4-8: qdb homo +
Dir: C16H18O-0 Parent bond 3-4: Qdb bond 2-3: qdb homo
Dir: C16H18O-0 Parent bond 3-35: Qdb bond 3-21: qdb homo
Dir: C16H16O2-0 Parent bond 4-5: Qdb bond 4-5: qdb homo
Dir: C16H18O-0 Parent bond 4-8: Qdb bond 4-8: qdb homo +
Dir: C16H16O2-0 Parent bond 5-9: Qdb bond 5-9: qdb homo +
Dir: C20H16O3-0 Parent bond 10-11: Qdb bond 10-15: qdb homo
Dir: C20H16O3-0 Parent bond 10-15: Qdb bond 10-15: qdb homo
Dir: C20H16O3-0 Parent bond 11-12: Qdb bond 14-15: qdb homo
Dir: C20H16O3-0 Parent bond 11-16: Qdb bond 15-19: qdb homo +
Dir: C20H16O3-0 Parent bond 12-13: Qdb bond 12-13: qdb homo
Dir: C20H16O3-0 Parent bond 12-17: Qdb bond 12-17: qdb homo +
Dir: C20H16O3-0 Parent bond 13-14: Qdb bond 12-13: qdb homo
Dir: C20H16O3-0 Parent bond 13-20: Qdb bond 13-20: qdb homo
Dir: C20H16O3-0 Parent bond 14-15: Qdb bond 14-15: qdb homo
Dir: C20H16O3-0 Parent bond 14-18: Qdb bond 12-17: qdb homo +
Dir: C20H16O3-0 Parent bond 15-19: Qdb bond 15-19: qdb homo +
Dir: C20H16O3-0 Parent bond 20-21: Qdb bond 20-21: qdb homo
Dir: C16H15NO5-0 Parent bond 20-22: Qdb bond 11-13: qdb homo -
Dir: C16H15NO5-0 Parent bond 22-23: Qdb bond 13-14: qdb homo
Dir: C16H15NO5-0 Parent bond 23-24: Qdb bond 14-15: qdb homo
Dir: C16H15NO5-0 Parent bond 23-28: Qdb bond 14-19: qdb homo
Dir: C16H15NO5-0 Parent bond 24-25: Qdb bond 15-16: qdb homo
Dir: C16H15NO5-0 Parent bond 24-29: Qdb bond 15-20: qdb homo +
Dir: C11H13NO5-0 Parent bond 25-26: Qdb bond 6-7: qdb homo -
Dir: C16H15NO5-0 Parent bond 25-32: Qdb bond 16-23: qdb homo
Dir: C11H13NO5-0 Parent bond 26-27: Qdb bond 7-8: qdb homo
Dir: C11H15NO4-0 Parent bond 26-36: Qdb bond 5-14: qdb homo -
Dir: C16H15NO5-0 Parent bond 27-28: Qdb bond 18-19: qdb homo
Dir: C16H15NO5-0 Parent bond 27-30: Qdb bond 18-21: qdb homo +
Dir: C16H15NO5-0 Parent bond 28-31: Qdb bond 19-22: qdb homo +
Dir: C16H15NO5-0 Parent bond 32-33: Qdb bond 23-24: qdb homo
Dir: C16H15NO5-0 Parent bond 32-34: Qdb bond 23-24: qdb homo
Dir: C12H18O-1 Parent bond 35-54: Qdb bond 11-12: qdb homo -
Dir: C11H15NO4-0 Parent bond 36-37: Qdb bond 14-15: qdb homo
Dir: C13H20O-0 Parent bond 37-38: Qdb bond 12-13: qdb homo -
Dir: C11H15NO4-0 Parent bond 37-39: Qdb bond 15-17: qdb homo +
Dir: C11H15NO4-0 Parent bond 37-81: Qdb bond 15-27: qdb homo
Dir: C13H20O-0 Parent bond 38-40: Qdb bond 13-15: qdb homo
Dir: C13H20O-0 Parent bond 38-41: Qdb bond 13-16: qdb homo +
Dir: C13H20O-0 Parent bond 38-42: Qdb bond 13-16: qdb homo +
Dir: C8H18O-0 Parent bond 40-43: Qdb bond 5-8: qdb homo -
Dir: C13H20O-0 Parent bond 40-44: Qdb bond 15-19: qdb homo +
Dir: C13H20O-0 Parent bond 40-45: Qdb bond 15-19: qdb homo +
Dir: C8H18O-0 Parent bond 43-46: Qdb bond 8-11: qdb homo +
Dir: C8H18O-0 Parent bond 43-47: Qdb bond 8-11: qdb homo +
Dir: C8H18O-0 Parent bond 43-48: Qdb bond 8-13: qdb homo
```

```
Dir: C8H18O-0 Parent bond 48-49: Qdb bond 13-14: qdb homo +
Dir: C8H18O-0 Parent bond 48-50: Qdb bond 13-14: qdb homo +
Dir: C8H18O-0 Parent bond 48-51: Qdb bond 13-16: qdb homo
Dir: C8H18O-0 Parent bond 51-52: Qdb bond 16-17: qdb homo +
Dir: C8H18O-0 Parent bond 51-53: Qdb bond 16-17: qdb homo +
Dir: C8H18O-0 Parent bond 51-85: Qdb bond 16-23: qdb homo
Dir: C12H18O-1 Parent bond 54-55: Qdb bond 12-13: qdb homo +
Dir: C12H18O-1 Parent bond 54-56: Qdb bond 12-13: qdb homo +
Dir: C12H18O-1 Parent bond 54-57: Qdb bond 12-15: qdb homo
Dir: C12H18O-1 Parent bond 57-58: Qdb bond 15-17: qdb homo +
Dir: C12H18O-1 Parent bond 57-59: Qdb bond 15-17: qdb homo +
Dir: C12H18O-1 Parent bond 57-60: Qdb bond 15-18: qdb homo
Dir: C12H18O-0 Parent bond 60-61: Qdb bond 15-19: qdb homo +
Dir: C12H18O-0 Parent bond 60-62: Qdb bond 15-19: qdb homo +
Dir: C7H16O-1 Parent bond 60-63: Qdb bond 8-11: qdb homo -
Dir: C8H18O-0 Parent bond 63-64: Qdb bond 8-11: qdb homo +
Dir: C8H18O-0 Parent bond 63-65: Qdb bond 8-11: qdb homo +
Dir: C8H18-0 Parent bond 63-66: Qdb bond 10-13: qdb homo -
Dir: C8H18-0 Parent bond 66-67: Qdb bond 13-14: qdb homo +
Dir: C8H18-0 Parent bond 66-68: Qdb bond 13-14: qdb homo +
Dir: C8H18-0 Parent bond 66-69: Qdb bond 10-13: qdb homo
Dir: C8H18-0 Parent bond 69-70: Qdb bond 13-14: qdb homo +
Dir: C8H18-0 Parent bond 69-71: Qdb bond 13-14: qdb homo +
Dir: C8H18-0 Parent bond 69-72: Qdb bond 10-13: qdb homo
Dir: C8H18-0 Parent bond 72-73: Qdb bond 13-14: qdb homo +
Dir: C8H18-0 Parent bond 72-74: Qdb bond 13-14: qdb homo +
Dir: C8H18-0 Parent bond 72-75: Qdb bond 7-10: qdb homo
Dir: C8H18-0 Parent bond 75-76: Qdb bond 16-17: qdb homo +
Dir: C8H18-0 Parent bond 75-77: Qdb bond 16-17: qdb homo +
Dir: C8H18-0 Parent bond 75-78: Qdb bond 4-7: qdb homo
Dir: C8H18-0 Parent bond 78-79: Qdb bond 4-5: qdb homo +
Dir: C8H18-0 Parent bond 78-80: Qdb bond 4-5: qdb homo +
Dir: C8H18-0 Parent bond 78-89: Qdb bond 19-22: qdb homo
Dir: C11H15NO4-0 Parent bond 81-82: Qdb bond 27-28: qdb homo +
Dir: C11H15NO4-0 Parent bond 81-83: Qdb bond 27-28: qdb homo +
Dir: C11H15NO4-0 Parent bond 81-84: Qdb bond 27-28: qdb homo +
Dir: C8H18O-0 Parent bond 85-86: Qdb bond 23-24: qdb homo +
Dir: C8H18O-0 Parent bond 85-87: Qdb bond 23-24: qdb homo +
Dir: C8H18O-0 Parent bond 85-88: Qdb bond 23-24: qdb homo +
Dir: C8H18-0 Parent bond 89-90: Qdb bond 22-23: qdb homo +
Dir: C8H18-0 Parent bond 89-91: Qdb bond 22-23: qdb homo +
Dir: C8H18-0 Parent bond 89-92: Qdb bond 22-23: qdb homo +
```

Appendix C

This appendix contains the source for the more ‘important’ portions of the software system. Not all programs are included, but the major ones are. They are organized by directory. The sections are titled by the directory name, and subtitled by the general purpose of the program, unless there is only one type of program in that directory, as outlined in Chapter 3. Oftentime, programs in one directory rely on libraries or routines in other directories. This would be made clear by included the config.pl from each directory, but for the sake of brevity, these programs have been omitted; they can be found either on the enclosed CD, or at the permanent home of the project.

Compile_all_fudge_scripts.pl

```perl
#!/home/radke/bin/perl/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net


use strict;

eval { require 5.6.1 }
  or die <<MESSAGE;
##################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
##################################################################
MESSAGE

# This script compiles and installs everything needed to begin development
# of the contained package.  Simply run it to set up the custom perl
# libraries, etc.  It will only run on *NIX systems (due to the use of
# `cwd`), but since it's a kludge, I don't care *smile*.  It also does
# no taint checking, since it blasts all over the place with the path.


my($initial_dir) = `pwd`;
chomp($initial_dir);

print "Building in $initial_dir/shlib/CFUNCS\n";
chdir("$initial_dir/shlib/CFUNCS");
system("perl ./Makefile.PL");
system("make realclean");
system("perl ./Makefile.PL");
system("make");
system("make test");
system("make install");

print "Building in $initial_dir/qdb\n";
chdir("$initial_dir/qdb");
system("./configure.pl");
system("make clean");
system("make");

print "Building in $initial_dir/qdb/qdb_maintenance_utilities/utilities\n";
chdir("$initial_dir/qdb/qdb_maintenance_utilities/utilities");
system("./configure.pl");
system("make clean");
system("make");

print "Building in $initial_dir/general\n";
chdir("$initial_dir/general");
print "Compiling chkmem\n";
system("rm -f ../qdb/chkmem");
system("cc -o chkmem chkmem.c");
system("ln -s $initial_dir/general/chkmem $initial_dir/qdb/chkmem");

print "Building in $initial_dir/genff\n";
chdir("$initial_dir/genff");
print "Compiling genff\n";
system("./configure.pl");
system("make clean");
system("make");

chdir($initial_dir);


print "All done with installations\n";

exit 0;
```

cmap

map_charges.pl

```perl
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# Note that this program is not written with the -T switch.  Since it
# receives no data from the outside world, we needn't be paranoid
# about it's security.  If a hacker has shell access, there's probably
# jucier targets than a force field development sub-program.  For
# details on the workings of this program, see README.txt in it's
# directory.  Aside from during development, the user will never
# directly call this program, but will instead be using a master
# program that uses this one to do some of it's work.

use strict;
use Socket qw(:DEFAULT :crlf);
use IO::Socket;

# The following use statment allows us to know where we were called
# from.  This is very important for being able to use the modules
# included in the distribution, which are in relative locations to
# this program.
use FindBin qw($RealBin);
my ($starting_path) = $RealBin;

eval { require 5.6.1 }
  or die <<MESSAGE;
##################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
##################################################################
MESSAGE

# Global (my) variable declarations
my($server_port);
my($server_host);
my($db_path);

# Begin processing .qdb_checkrc file
require "$starting_path/../general/rc_file_handling.pl";

open("RCFILE", "<$starting_path/../qdb/.qdb_checkrc") or die
  "Unable to open .qdb_checkrc ... exiting\n";

$server_host = read_scalar("RCFILE", "query_server_host");
defined($server_host) or die
  "Unable to find server_host in .qdb_checkrc file ... exiting\n";

$server_port = read_scalar("RCFILE", "query_server_port");
defined($server_port) or die
  "Unable to find server_port in .qdb_checkrc file ... exiting\n";

$db_path = read_scalar("RCFILE", "db_path");
defined($db_path) or die
  "Unable to find db_path in .qdb_checkrc file ... exiting\n";

close("RCFILE");

until (<STDIN> =~ /^Begin atom map list:$/) {
}

my(@charge_map) = ();
my($line);
my($i);

until ( ($line = <STDIN>) =~ /^End atom map list:$/) {

  # We need to verify that this is not from an incomplete request
  unless ( $line =~
       /^Dir: ([\w]+-[\d]+) Parent atom [\d]+: Qdb atom ([\d]+): qdb[\s]*$/) {
  print STDERR "The input we received was not in the expected " .
    "format.  Specifically: \"$line\".  This line may be " .
    "from an incomplete run of the force field generator" and
      die "Incorrect format, see STDERR for details\n";
  }

  # And get the relevant details, we make our queries on the map later.
  push(@charge_map, [$1, $2]);

}
```

```
# Uncomment the following to look at the atom map list.
#foreach (@charge_map) {
#  print join("\t", @{$_}) . "\n";
#}

# print "And the indexed version of this is:\n";
#foreach $i (1..$#charge_map) {
#  print "get charge chelpg $charge_map[$i][0] $charge_map[$i][1] ";
#  if (-e "/private_ffd/qdb/$charge_map[$i][0]/charges.chelpg") {
#  print "Charge file found\n";
#  } else {
#  print "Error!\n";
#  }

#sleep(1);
#}

# Set up the client
my($remote) = IO::Socket::INET->new( Proto => "tcp",
                   PeerAddr => "$server_host:$server_port", Type =>
SOCK_STREAM);

if (!$remote) {
  die "Unable to create connection to server";
}
$remote->autoflush(1);

# Set the end of line character to read input from an internet server
my($oldsep) = $/;
$/ = $CRLF;

foreach $i (0..$#charge_map) {

  print $remote "get charge chelpg $charge_map[$i][0] $charge_map[$i][1]$CRLF";
  $charge_map[$i] = <$remote>;
  chomp($charge_map[$i]);
}

$/ = $oldsep;

foreach $i (0..$#charge_map) {
  print $charge_map[$i] . "\n";
}

my($sum) = 0;
$i = 1;

foreach (@charge_map) {
  $sum += $_;
  $i++;
}

# Now we offset all of the charges, so the total charge is 0.  Once
# again, this is a workaround, and this program _should_ take an
# argument for what the total charge is supposed to be.

my $offset = $sum / $i;

foreach (@charge_map) {
  $_ -= $offset
}

foreach $i (0..$#charge_map) {
  print $charge_map[$i] . "\n";
}


$sum = 0;
$i = 1;

foreach (@charge_map) {
  $sum += $_;
  $i++;
}

# Uncomment the following to see the error of the total division
# print "The new sum of charges on $i atoms is $sum\n";

exit(0);
```

finstr

prepfinal.ff

```
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# The purpose of this program is to extract all of the necessary
# information from the database, and initialize the data structures.
# It can then be dumped (using the Data::dumper) module to a file that
# can easily be reconstructed by any translator.  Note that due to
# time constraints, this program will read all of its information from
# the quantum database, therefore 'breaking' the client/server model
# that has been adhered to until now (4-11-2002).  This should
# _definately_ be fixed in a future re-write.

use strict;

# The following use statment allows us to know where we were called
# from.  This is very important for being able to use the modules
# included in the distribution, which are in relative locations to
# this program.
use FindBin qw($RealBin);
my ($starting_path) = $RealBin;

eval { require 5.6.1 }
  or die <<MESSAGE;
#################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
#################################################################
MESSAGE

# Global (my) variable declarations
my($db_path);
my($i);

# Begin processing .qdb_checkrc file
require "$starting_path/../general/rc_file_handling.pl";

open("RCFILE", "<$starting_path/../qdb/.qdb_checkrc") or die
  "Unable to open .qdb_checkrc ... exiting\n";

$db_path = read_scalar("RCFILE", "db_path");
defined($db_path) or die
  "Unable to find db_path in .qdb_checkrc file ... exiting\n";

close("RCFILE");

# Since this program needs to be called with a filename for input (so
# it knows what to send to the map_charges.pl program), we will
# require an argument.

if ($#ARGV != '0') {
  die "Please provide exactly 1 argument, which is the filename of the " .
    "input file.  The input file should come from either qdb_check, or " .
    "qdb_input_server.pl";
}

open ('INFILE', "<$ARGV[0]") or
  die "Unable to open $ARGV[0] for reading, cannot continue";


# We need to read many parameters, the following should be a (growing)
# complete list of all of the information that needs to be retrieved
# from the database.
# Parent structure (labels, coordinates, charges, connectivity,
# qcodes, and stereochemistry)
# Same information for all children, they will be referenced by their
# qdb directory name.  The children do need charges, since we're
# handling that mapping ourselves.  They also need qcodes for atom typing
# Atom map list
# Bond map list
my %parent_molecule;
my %children;
my @atom_map_list;
my %bond_map_list;

# All of this information will be encoded as 0 base, with no implicit
# hydrogens.  If a particular application (such as Matt's force field
# program) wishes to do other things with the data, that program will
# be called to process the dump from this program.

# Finally, note that this program takes its input from INFILE (The
# input is a 'finished' output from qdb_local_submit.pl or qdb_check).
# It also prints all of it's (dumped) data to STDOUT, which should be
# saved to a file, or piped into the translator.  As usual, it will be
# very picky about the input being correctly formatted, so mistakes
# don't go unnoticed.

# The connectivity will be a hash of atoms, and a list of what atoms
# they're bonded to, with what bond orders.

my $line = <INFILE>;
my $expectval = "Begin parent molecule:";

unless ($line =~ /^$expectval/ ) {
  die "Bad input:\nExpected:  $expectval\nReceived:  $line";
}

$line = <INFILE>;
$expectval = "Begin coordinates";

unless ($line =~ /^$expectval/ ) {
  die "Bad input:\nExpected:  $expectval\nReceived:  $line";
}

# Now we should be at the parent coordinates, start reading them.

$expectval = "Begin connectivity";
$line = <INFILE>;
```

```
chomp $line;

until ($line =~ /^$expectval/ ) {

  my($label, $x, $y, $z) = split(",", $line);

  push(@{$parent_molecule{labels}}, $label);
  push(@{$parent_molecule{coordinates}}, [$x, $y, $z]);

  $line = <INFILE>;
  chomp $line;
}

unless ($line =~ /^$expectval/ ) {
  die "Bad input:\nExpected:  $expectval\nReceived:  $line";
}

# Get the connectivity.  This section will be a bit differenct, since
# we're building a hash that says (for each atom) what points to what,
# and with what bond orders.
my %scratch_hash;

$expectval = "Begin qcodes";
$line = <INFILE>;
chomp $line;

until ($line =~ /^$expectval/ ) {

  my($atom1, $atom2, $bond_order) = split(" ", $line);

  push(@{$scratch_hash{$atom1}}, [$atom2, $bond_order]);
  push(@{$scratch_hash{$atom2}}, [$atom1, $bond_order]);

  $line = <INFILE>;
  chomp $line;
}

# The { %scratch_hash } means give me an anonymous reference to that
# hash, in effect, copying the values.  Note that the lists within the
# hash were also made as anonymous references.
$parent_molecule{connectivity} = { %scratch_hash };

# Initialize the qcodes.  The qcodes will (presumably) only be used to
# determine exact matches for typing in subsequent force fields.
# Thus, they will be packed as a hash whose key is the qcode itself,
# and whose value is a reference to a list of atoms who contain that
# qcode.
%scratch_hash = ();


$line = <INFILE>;
chomp $line;

$i = 0;
until ($line =~ /^Begin stereochemical descriptors/  or
     $line =~ /^End molecule output/ ) {

  push(@{$scratch_hash{$line}}, $i);

  $line = <INFILE>;
  chomp $line;
  $i++;
}

# It would be easy here to convert the keys to shorter values, but at
# some point in the future, a program may wish to do something with
# the qcodes, so they're left in place.
$parent_molecule{qcodes} = { %scratch_hash };

# If we need to initialize the stereochemical descriptors, do so here.
if ($line =~ /^Begin stereochemical descriptors/) {
  $expectval = "End molecule output";
  $line = <INFILE>;
  chomp $line;

  until ($line =~ /^$expectval/ ) {

  my($atom, $descriptor) = split(" ", $line);
  push(@{$parent_molecule{stereochemistry}}, [ $atom, $descriptor ]);

  $line = <INFILE>;
  chomp $line;
  }
}

$line = <INFILE>;
$expectval = "Begin atom map";

unless ($line =~ /^$expectval/ ) {
  die "Bad input:\nExpected:  $expectval\nReceived:  $line";
}

# Ok, we're ready to read the atom map list.  The atom map list will
# be an ordered list of references to a hash.  The hash will have 2
# keys; directory, and fragatom.

$expectval = "End atom map list";
$line = <INFILE>;
chomp $line;

until ($line =~ /^$expectval/ ) {

  $line =~ /Dir: ([\w-]+) Parent atom (\d+): Qdb atom (\d+): qdb/ or
  die "Badly formatted atom map line.\nExpected (regex): " .
      '/Dir: ([\w-]+) Parent atom (\d+): Qdb atom (\d+): qdb/' .
      "Received: $line";

  $atom_map_list[$2] = {'directory' => $1, 'fragatom' => $3 };

  $line = <INFILE>;
  chomp $line;
}

# Now we can initialize the bond map list.  The format will be quite
# similar, except the bonds will be longer strings, such as '25-32'.
# These can easily be split later if it is required.  Since the keys
# will now be strings, we need to use a hash instead.
$expectval = "Begin bond map list";
$line = <INFILE>;
chomp $line;

unless ($line =~ /^$expectval/ ) {
  die "Bad input:\nExpected:  $expectval\nReceived:  $line";
}

$expectval = "End bond map list";
$line = <INFILE>;
chomp $line;

until ($line =~ /^$expectval/ ) {

  $line =~
/Dir: ([\w-]+) Parent bond ([\d-]+): Qdb bond ([\d-]+): qdb (homo|enantio)/
  or die "Badly formatted atom map line.\nExpected (regex): " .
        '/Dir: ([\w-]+) Parent bond ([\d-]+): Qdb bond ([\d-]+): qdb ' .
       '(homo|enantio)/' .
        "Received: $line";

  if ($4 eq 'enantio') {
  die "We don't know how to handle enantiomeric fragments yet.";
  }

  $bond_map_list{$2} = {'directory' => $1, 'fragbond' => $3 };

  $line = <INFILE>;
  chomp $line;
}

close('INFILE');

# Finally, we need to get the charges into a list and attach them to
# our parent molecule.

open(CHARGES, "$starting_path/../cmap/map_charges.pl < $ARGV[0] |" ) or
  die "Unable to execute $starting_path/../cmap/map_charges.pl < $ARGV[0] |";

my @chargelist = <CHARGES>;
chomp(@chargelist);

close(CHARGES);

$parent_molecule{charges} = [ @chargelist ];


# Ok, everything from the input file is finalized.  The last step is
# to loop through the directories in the database, and initialize each
# of the fragments with the same values as we did with the parent fragment.

my %dirhash;


foreach (@atom_map_list) {
  my (%entry) = %{$_};
  $dirhash{$entry{directory}} = 1;
}

foreach (keys(%bond_map_list)) {
  $dirhash{${$bond_map_list{$_}}{directory}} = 1;
}

# For each of the directories, we need to format the labels, coordinates,
# connectivity, stereochemistry, qcodes, and charges (symmetrized).
foreach (keys(%dirhash)) {
  my %fragment = ();

  my $dir = $_;

  # Get geometry
  open(STRUCT, "<$db_path/$dir/Original_structure.raw") or
  die "Unable to open <$db_path/$_/Original_structure.raw for reading";

  while (<STRUCT>) {
  chomp;
  my($label, $x, $y, $z) = split(",", $_);

  push(@{$fragment{labels}}, $label);
  push(@{$fragment{coordinates}}, [$x, $y, $z]);
  }

  close(STRUCT);

  # Get connectivity
  open(CONN, "<$db_path/$dir/Connectivity.raw") or
  die "Unable to open <$db_path/$dir/Connectivity.raw for reading";

  my (%scratch_hash) = ();
  while (<CONN>) {
  chomp;
  my($atom1, $atom2, $bond_order) = split(" ", $_);

  push(@{$scratch_hash{$atom1}}, [$atom2, $bond_order]);
  push(@{$scratch_hash{$atom2}}, [$atom1, $bond_order]);

  }
  close(CONN);

  $fragment{connectivity} = { %scratch_hash };

  # We may or may not have stereochemistry to record.
  open(STER, "<$db_path/$dir/Stereochemical_descriptors") or
  die "Unable to open <$db_path/$_/Stereochemical_descriptors for reading";

  while (<STER>) {
  chomp;
  my($atom, $descriptor) = split(" ", $_);
  push(@{$fragment{stereochemistry}}, [ $atom, $descriptor ]);
  }
  close(STER);
```

```
  # Get qcodes, format them the same as in the parent molecule
  open(QCODES, "<$db_path/$dir/Qcodes") or
  die "Unable to open <$db_path/$dir/Qcodes for reading";

  %scratch_hash = ();
  $i = 0;
  while (<QCODES>) {
  chomp;

  push(@{$scratch_hash{$_}}, $i);

  $i++;
  }
  close(QCODES);

  $fragment{qcodes} = { %scratch_hash };

  # Finally, we need to retrieve the charges, and symmetrize them in
  # an extra step, since we're not using the map charges program to do
  # so.  Once again, we take the shortcut of assuming it's the chelpg
  # charges.

  my(@chargelist);
  open(CHARGES,"<$db_path/$dir/charges.chelpg") or
  die "Unable to open <$db_path/$dir/charges.chelpg for reading";
  @chargelist = <CHARGES>;
  close(CHARGES);
  chomp(@chargelist);

  # Symmetrize the values in @chargelist.

  # I know the next line simply reiterates what happened a few lines
  # ago, but I want to keep this close to the rest of the code.
  %scratch_hash = %{$fragment{qcodes}};

  foreach (keys(%scratch_hash)) {
  my @atoms = @{$scratch_hash{$_}};

  # Foreach of these atoms, get an average value of charge, and
  # assign that average value to each of them.
  my($sum) = 0;
  foreach (@atoms) {
    $sum += $chargelist[$_];
  }
  my $average = $sum / scalar(@atoms);
  foreach (@atoms) {
    $chargelist[$_] = $average;
  }
  }

  # Just in case we introduced a bias symmetrizing the charges, reset
  # them so they add up to 0.
  my $sum = 0;
  foreach (@chargelist) {
  $sum += $_;
  }
  my $offset = $sum / scalar(@chargelist);
  foreach (@chargelist) {
  $_ -= $offset;
  }

  # And give the chargelist to this fragment.
  $fragment{charges} = [ @chargelist ];


  # That's it, this fragment is initialized, put it on the children hash
  $children{$dir} = { %fragment };
}

# Ok, we've now finally initalized all of the data completely, we will
# simply dump it to stdout, so it can be restored by whatever filter
# wants it.

use Data::Dumper;  # Don't worry about this use statement being late,
                   # all use statements are evaluated before execution

$Data::Dumper::Indent = 2;  # Make 'pretty' dumped data
$Data::Dumper::Purity = 1;  # Catch any self-referential data.  This
                            # may not be necessary for our
                            # application, but performance is not an
                            # issue.

my $saved = Data::Dumper->Dump(
                [
               \%parent_molecule, \@atom_map_list,
               \%bond_map_list, \%children
                ],
                [
               '*parent_molecule', '*atom_map_list',
               '*bond_map_list', '*children'
                ]
               );

# All of the entire data structure is now saved in $saved.  We simply
# print it out, and exit;

print $saved;

exit(0);



#######################################################################
#  End normal program, begin example access code                      #
#######################################################################

# The following is code to print out the contents of the data

# Look at the parent coordinates
print "Outputting coordinates\n";
for ($i = 0; $i <= $#{$parent_molecule{labels}}; $i++) {
  print "${$parent_molecule{labels}}[$i]," .
      "${$parent_molecule{coordinates}}[$i][0]," .
      "${$parent_molecule{coordinates}}[$i][1]," .
      "${$parent_molecule{coordinates}}[$i][2]\n";
}

# Look at the parent connectivity.  I know the choice of packing makes
# things a little tricky, but the data is preserved in a single list.
%scratch_hash = %{$parent_molecule{connectivity}};
foreach (sort {$a <=> $b} keys(%scratch_hash)) {
  my (@thisbonds) = ( @{$scratch_hash{$_}});
  print "$_:  ";
  foreach (@thisbonds) {
  print join("=>", @{$_}) . " "
  }
  print "\n";
}

# Look at the qcode information
%scratch_hash = %{$parent_molecule{qcodes}};
print "Atoms with given qcodes:\n";
$i = 0;
foreach (keys(%scratch_hash)) {
  print "$i:  " . join(", ", @{$scratch_hash{$_}}) . "\n";
  $i++;
}

# Look at any stereochemistry available
foreach (@{$parent_molecule{stereochemistry}}) {
  print join(", ", @{$_}) . "\n";
}

# Look at the parent molecule's charges
print join(", ", @{$parent_molecule{charges}}) . "\n";

# Look at the atom map list
for ($i = 0; $i <= $#atom_map_list; $i++) {
  print "Atom $i uses fragment atom ";
  print "${$atom_map_list[$i]}{fragatom} ";
  print "in directory ${$atom_map_list[$i]}{directory}\n";
}

# Look at the bond map list
foreach ( keys(%bond_map_list) ) {
  print "Bond $_ uses fragment bond ";
  print "${$bond_map_list{$_}}{fragbond} ";
  print "in directory ${$bond_map_list{$_}}{directory}\n";
}



# print "The required directories follow:\n";
# foreach (keys(%dirhash)) {
#   print "$_\n";
# }
# print "End required directories\n";

# Following is a demonstration of how to look at all of the
# information on all of the children.
# Here we test the printing of the children;
foreach (keys(%children )) {
  my $dir = $_;

  print "Outputting coordinates for directory $dir\n";
  for ($i = 0; $i <= $#{$children{$dir}{labels}}; $i++) {
  print "${$children{$dir}{labels}}[$i]," .
    "${$children{$dir}{coordinates}}[$i][0]," .
    "${$children{$dir}{coordinates}}[$i][1]," .
    "${$children{$dir}{coordinates}}[$i][2]\n";
  }

  print "Connectivity of fragment follows\n";
  %scratch_hash = %{$children{$dir}{connectivity}};
  foreach (sort {$a <=> $b} keys(%scratch_hash)) {
  my (@thisbonds) = ( @{$scratch_hash{$_}});
  print "$_:  ";
  foreach (@thisbonds) {
    print join("=>", @{$_}) . " "
  }
  print "\n";
  }

  print "Stereochemistry follows\n";
  foreach (@{$children{$dir}{stereochemistry}}) {
  print join(", ", @{$_}) . "\n";
  }
  print "Stereochemical information finished\n";
}
```

makestr.pl

```
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# The purpose of this program is to format all of the data collected
# and organized in prepfinalff.pl, and to apply it to create exactly
# the structure Matt needs for his programs to run.  This should
# minimize the effort involved in that particular task.

# Function prototypes;
sub assign_unique_names(\%);
sub attach_str_file(\%);
sub absorb_hydrogens(\%\%);
sub dont_absorb_hydrogens(\%\%);
sub make_com_file(\%);
sub attach_mff_file(\%);
sub attach_frozen_keys(\%);
sub make_dir_structure;
sub map_types_to_children;
sub map_fragbond_to_parent;
sub map_fragangles_to_parent;
sub configure_torsions;
sub find_max_symmetry(\%$$);
sub prompt($\@;$);
sub enter_interactive;
sub fit_all_torsions;

# The following use statment allows us to know where we were called
# from.  This is very important for being able to use the modules
# included in the distribution, which are in relative locations to
# this program.

use FindBin '$RealBin';

my ( $starting_path ) = $RealBin;


BEGIN {
  # Since our own modules aren't properly installed, add to the INC
  # list at compile time.  Note that $starting_path won't be available
  # until the execution of the program begins.
  push(@INC, "$RealBin/../perl_modules");
}

use LINALG ':basic';

eval { require 5.6.1 }
  or die <<MESSAGE;
###################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
###################################################################
MESSAGE

# Global (my) variable declarations
my($db_path);
my($i);

# Before we start with any of the program, process command line
# options.  After the first run of this program we found we needed to
# also be able to generate an explicit hydrogen model.  We'll simply
# add an option to do this.  This command line handling is taken
# primarily from ../qdb/qdb_maintenence_utilities/qdb_utilities.pl
# Special variables used only in dealing with command line options:
my($explicit_h) = 0;

my($help_flag) = 0;

my($all_options) = q/
  "explicit|x" =>  \$explicit_h,    # Use explicit H's.
  "help|h|?"    =>  \$help_flag,
/;

use Getopt::Long;
Getopt::Long::Configure( qw/no_ignore_case_always bundling/ );
my($cmd_line) = GetOptions ( eval($all_options) );

# First order of business is to see if help was requested.  If so, simply
# print out how Getopt::Long was called.  This isn't beautiful, but it
# requires the least maintenance while options are added.
if ($help_flag) {
  print <<HELP_MSG;
Usage:
  $0 [x]

  Outputting Getopt::Long configuration.  See
  http://www.perldoc.com/perl5.6/lib/Getopt/Long.html for more
  information.
HELP_MSG

  print $all_options;
  print <<HELP_MSG;

Nothing done
HELP_MSG

  exit 1;
}

# Begin processing .qdb_checkrc file
require "$starting_path/../general/rc_file_handling.pl";

open("RCFILE", "<$starting_path/../qdb/.qdb_checkrc") or die
  "Unable to open .qdb_checkrc ... exiting\n";

$db_path = read_scalar("RCFILE", "db_path");
defined($db_path) or die
  "Unable to find db_path in .qdb_checkrc file ... exiting\n";

my($local_ab_initio_program) =
  read_scalar("RCFILE", "local_ab_initio_program");
defined($local_ab_initio_program) or die
  "Unable to find local_ab_initio_program in .qdb_checkrc file " .
  "... exiting\n";

close("RCFILE");

# And load the appropriate ab_initio program specific function names
require("$starting_path/../perl_modules/" .
    "${local_ab_initio_program}_functions.pl");

# Extracted prototypes from g98_functions.pl.  That file should
# eventually be ported to a 'modern module', so prototypes are used
# when we 'use' the module.
sub get_last_dihedral_and_energy($\@);
sub get_optimized_structure($);
sub find_end_of_optimization (*);
sub atomic_number_to_label($);
sub hartree_to_kcal_per_mole($);
sub make_dihed_inp_file($$\@\@$);
sub read_first_geometry($);
sub extract_chelpg_charges($);

# The following values must be declared before we can initialize them
# from the final file created by prepfinalff.pl
my %parent_molecule;
my %children;
my @atom_map_list;
my %bond_map_list;

# Since this program needs to be called with a filename for input (so
# it can do user prompting), we do that now.

if ($#ARGV != '0') {
  die "Please provide exactly 1 argument, which is the filename of the " .
    "input file.  The input file should come from either qdb_check, or " .
    "qdb_input_server.pl";
}

open (INFILE, "<$ARGV[0]") or
  die "Unable to open $ARGV[0] for reading, cannot continue";

# All of this information is encoded base 0, with no implicit
# hydrogens, or other fancy work done.  This program is responsible
# for doing the 'fancy work'

my $oldsep = $/;
$/ = undef;
eval <INFILE>;
close INFILE;
$/ = $oldsep;

# Note that some recent changes to how qdb_query_server.pl works
# mandates that we 'mangle' the bond map list, so it reports torsion
# directories with the convention that the two atoms defining the
# torsion are in numerical order.  We do this now.
foreach (keys %bond_map_list) {
  my ($atom1, $atom2) = split(/-/, $bond_map_list{$_}{fragbond});
  if ($atom1 > $atom2) {
  $bond_map_list{$_}{fragbond} = "$atom2-$atom1";
  }
}

# This variable controls whether we wipe out the entire old directory
# structure, or simply overwrite the files that are there.


my (@allowed_answers) = qw /o r s t q/;
my $prompt = <<MSG;
o)  Overwrite the directory structure and initialize
r)  Refresh the directory structure without destroying existing files
s)  Skip all initialization, and go into interactive mode immediately
t)  Try to fit and verify all torsions, this option is dangerous, and
    will definately take some time.  It will also _not_ initialize the
    directory structure, so you should refresh or overwrite if you're
    not certain the directories are properly set up.
q)  Quit before doing anything
What shall we do?
MSG
chomp $prompt;

my $response =
  prompt($prompt, @allowed_answers, 's' );
if ($response eq 'q') {
  print "Exiting\n";
  exit;
} elsif ($response eq 'o') {
  $cleandir = 1;
} elsif ($response eq 's') {
  # In this case, we go directly to interactive mode, and exit when
  # done.  Since we're doing this, we need to make sure interactive
  # mode does not rely on any of the initialized data.
  chdir 'myff' or die "Could not change directory to myff";
  enter_interactive;
  exit;
} elsif ($response eq 't') {
  fit_all_torsions;
} else {
  $cleandir = 0;
}

# Print statements will be scattered throughout the main body to
# provide status updates.
$| = 1;
print 'Initialization progress: \......../' . "\n";
print '                          \.';
```

```
# Unlike what I had previously thought, every single one of the
# fragments must have an 'almost complete' force field file to go with
# it, as well as an .str file.  This isn't a problem, we'll just need
# to keep the mapping information readily available.

# Another task is to assign unique Character strings to each atom.
# This will be done by reserving the first 2 characters for the atom
# label, and simply providing a serial number after that.  Let's do
# that task now: Note that we must call assign_unique_names on the
# parent first, since the function keeps a 'blacklist' of taken names,
# and we will later assign the names from the parent to the proper
# fragment atoms.

assign_unique_names(%parent_molecule);

# Now do the same for all of the fragments.  Important lesson: When
# manipulatin the children hash (adding things to it) we _must_ pass
# the original hash to the function, or the new 'types' portion of the
# hash is simply copied into the copy, not the actual children.  Just
# use the following loop for a template.
foreach (keys( %children )) {
  assign_unique_names( %{$children{$_}} );
}

# Now that we have unique names for all of our types of atoms, we need
# to create the absorbed molecules, both for the parent, and for all
# of the children.  Upon absorbing hydrogens, add a 'tosister' section
# to both hashes which maps the atoms in the current hash to the atoms
# in the absorbed hash.  Also, make the absorbed atoms have the charge
# of the sum of the original atom plus the absorbed hydrogens.
# Finally, change the type of the absorbed atom to C_1, C_2, or C_3,
# depending on how many hydrogens we absorbed.
my %abs_parent;
my %abs_children;
if ($explicit_h) {
  dont_absorb_hydrogens(%parent_molecule, %abs_parent);
} else {
  absorb_hydrogens(%parent_molecule, %abs_parent);
}

foreach (keys( %children )) {
  if ($explicit_h) {
  dont_absorb_hydrogens(%{$children{$_}}, %{$abs_children{$_}});
  } else {
  absorb_hydrogens(%{$children{$_}}, %{$abs_children{$_}});
  }
}

# Attach the frozen keys to the absorbed children.
attach_frozen_keys( %abs_children );

print '.';

# Ok, we need to use the atom types from the parent on all of the
# children.  This will require looking at both the 'tosister' maps,
# and looking at our very own @atom_map_list and %bond_map_list.  As
# it turns out, this step is not necessary, and in fact, is not done
# properly since the attached .mff files no longer end up with the
# proper labels, particularly in the torsions

#map_types_to_children;


# Create str files for all of our members
attach_str_file( %parent_molecule );
attach_str_file( %abs_parent );

foreach (keys( %children )) {
  attach_str_file( %{$children{$_}} );
}

foreach (keys( %abs_children )) {
  attach_str_file( %{$abs_children{$_}} );
}

print '.';

# We have str_files, we now need to attach a .mff (for Matt's force
# field) file to each of the absorbed fragments.  Since we'll only be
# using the absorbed fragments for mff's, we can leave the parents
# alone.  These will be attached as a new key under 'mff'.

attach_mff_file( %abs_parent );

# According to Matt's latest (Morning, 4/15/02) e-mail, the children
# do not require .mff files.  Regardless, we will still need force
# field parameters for every single atom type, and since the children
# will have some parameters that the parent doesn't we'll generate
# this file so we can extract them later.
foreach (keys( %abs_children )) {
  attach_mff_file( %{$abs_children{$_}} );
}

# We want the absorbed parent to have the bond lengths and angles that
# the optimized children did, thus the following two functions.
map_fragbond_to_parent;
map_fragangles_to_parent;

print '.';

# We need to attach enough information about the torsions that we can
# manipulate them easily in order to provide Matt's torsion fitting
# programs to run.  We will attach a torsions key to each member of
# %t_frags that has data available (and requested) for it in the
# database.  We'll normalize the energies to 0, and we'll also mark
# which four atoms (both in index and type) represent the torsion for
# each of the energy files provided.  Note that all potential torsions
# will be representted.  Finally, if we feel ambitions, we can try to
# determine if the torsion should be even or odd.

configure_torsions;

# Tasks: Write out complete directory structure, with appropriate
# files in all places.  After we do this, go into an interactive mode.
# After I've worked with Matt's fitting, I'll simply incorporate the
# syntax into this program.

print '.';

make_dir_structure;  # This function needs to create the input files
                     # for the actual torsion fitting program.  Matt
                     # sent me a sample, but it may or may not be out
                     # of date.

# After making the directory structure, and once again if/when we're
# feeling ambitious we'll enter an interactive mode for doing the
# actual torsion fitting.  This will make things a bit easier, since
# we will easily be able to slide between the relvant directories, etc.


# The following will dump all of our data, so it can be restored
# later.  It currently (morning 4/16/02) creates a file about 3.2MB.
# This file compresses to about 330k.
#$saved = Data::Dumper->Dump(
#              [
#               \%abs_children, \%children,
#               \%t_frags, \%parent_molecule,
#               \%abs_parent, \@atom_map_list,
#               \%bond_map_list,
#              ],
#              [
#               '*abs_children', '*children',
#               '*t_frags', '*parent_molecule',
#               '*abs_parent', '*atom_map_list',
#               '*bond_map_list',
#              ]
#               );

# print $saved;

print "./\n";

enter_interactive;

exit(0);




#########################################################################
#  End normal program, begin example access code                        #
#########################################################################

# The following is code to print out the contents of the data

# Look at the parent coordinates
print "Outputting coordinates\n";
for ($i = 0; $i <= $#{$parent_molecule{labels}}; $i++) {
  print "${$parent_molecule{labels}}[$i]," .
      "${$parent_molecule{coordinates}}[$i][0]," .
      "${$parent_molecule{coordinates}}[$i][1]," .
      "${$parent_molecule{coordinates}}[$i][2]\n";
}

# Look at the parent connectivity.  I know the choice of packing makes
# things a little tricky, but the data is preserved in a single list.
%scratch_hash = %{$parent_molecule{connectivity}};
foreach (sort {$a <=> $b} keys(%scratch_hash)) {
  my (@thisbonds) = ( @{$scratch_hash{$_}});
  print "$_:  ";
  foreach (@thisbonds) {
  print join("=>", @{$_}) . " "
  }
  print "\n";
}

# Look at the qcode information
%scratch_hash = %{$parent_molecule{qcodes}};
print "Atoms with given qcodes:\n";
$i = 0;
foreach (keys(%scratch_hash)) {
  print "$i:  " . join(", ", @{$scratch_hash{$_}}) . "\n";
  $i++;
}

# Look at any stereochemistry available
foreach (@{$parent_molecule{stereochemistry}}) {
  print join(", ", @{$_}) . "\n";
}

# Look at the parent molecule's charges
print join(", ", @{$parent_molecule{charges}}) . "\n";

# Look at the atom map list
for ($i = 0; $i <= $#atom_map_list; $i++) {
  print "Atom $i uses fragment atom ";
  print "${$atom_map_list[$i]}{fragatom} ";
  print "in directory ${$atom_map_list[$i]}{directory}\n";
}

# Look at the bond map list
foreach ( keys(%bond_map_list) ) {
  print "Bond $_ uses fragment bond ";
  print "${$bond_map_list{$_}}{fragbond} ";
  print "in directory ${$bond_map_list{$_}}{directory}\n";
}



# print "The required directories follow:\n";
# foreach (keys(%dirhash)) {
#   print "$_\n";
# }
# print "End required directories\n";
```

```
# Following is a demonstration of how to look at most of the
# information on all of the children. (The children have had data
# added to them, see the parent section for deatils on displaying that
# information.
foreach (keys(%children )) {
  my $dir = $_;

  print "Outputting coordinates for directory $dir\n";
  for ($i = 0; $i <= $#{$children{$dir}{labels}}; $i++) {
  print "${$children{$dir}{labels}}[$i]," .
    "${$children{$dir}{coordinates}}[$i][0]," .
    "${$children{$dir}{coordinates}}[$i][1]," .
    "${$children{$dir}{coordinates}}[$i][2]\n";
  }

  print "Connectivity of fragment follows\n";
  %scratch_hash = %{$children{$dir}{connectivity}};
  foreach (sort {$a <=> $b} keys(%scratch_hash)) {
  my (@thisbonds) = ( @{$scratch_hash{$_}});
  print "$_:  ";
  foreach (@thisbonds) {
    print join("=>", @{$_}) . " "
  }
  print "\n";
  }

  print "Stereochemistry follows\n";
  foreach (@{$children{$dir}{stereochemistry}}) {
  print join(", ", @{$_}) . "\n";
  }
  print "Stereochemical information finished\n";
}

# To see the types for all of the children:
foreach (keys( %children )) {
  my $dir = $_;
  # Uncomment the following to display the contents of this hash
  # Look at the qcode information

  my %fragment = %{$children{$dir}};

  print "For fragment $dir:\n";

  my %scratch_hash = %{$fragment{types}};
  print "Atoms with given types:\n";
  foreach (keys(%scratch_hash)) {
  print "$_:  " . join(", ", @{$scratch_hash{$_}}) . "\n";
  }

}

# To see sample absorbed fragments for all of the children, place the
# following snippet in main
foreach ( keys %abs_children ) {
  my $dir = $_;
  my %fragment = %{$abs_children{$dir}};
  print "Creating com file for $dir\n";

  open(COM, ">$dir.com") or die "Unable to open $dir.com for writing";
  select COM;
  make_com_file( %fragment );
  close COM;
  select STDOUT;
}

# To see sample mff and str files for all of the absorbed children,
# place the following snippet in main.
foreach ( keys %abs_children ) {
  my $dir = $_;
  my %fragment = %{$abs_children{$dir}};
  print "Creating str and mff files for $dir\n";

  open(COM, ">$dir.str") or die "Unable to open $dir.str for writing";
  select COM;
  print $fragment{str};
  close COM;

  open(COM, ">$dir.mff") or die "Unable to open $dir.mff for writing";
  select COM;
  print $fragment{mff};
  close COM;
  select STDOUT;
}



######################################################################
#  Begin functions.  Since this is simply a fast prototyping
#  situation, they will mainly be convenience functions, i.e., the
#  code is consolidated in functions instead of being centralized to
#  the main program flow.
######################################################################

# This function simply renames the keys in the qcodes hash, to reflect
# more 'typical' names for the atom types.  This will add a section to
# the hash which is called (aptly enough) 'types'.  Note that we
# cannot re-use any names for the children, so we will keep an active
# blacklist here.
BEGIN {

  my %taken_names;

  sub assign_unique_names(\%) {

  my $molref = shift;
  my %this_mol = %{$molref};

  my %names_index;
  my %types;

  %scratch_hash = %{$this_mol{qcodes}};
  foreach (keys(%scratch_hash)) {
  my $type;
  my @scratch_list;
  my $replabel = ${$this_mol{labels}}[${$scratch_hash{$_}}[0]];
  my $index = 0;

  if (length($replabel) == 1) {
  $type = "$ {replabel}_";
  } else {
  @scratch_list = split('', $replabel);
  $type = "$scratch_list[0]$scratch_list[1]";
  }

  # Now we check the taken_names index for an entry of this name,
  # if it doesn't exist, set the rest of the name to that value,
  # if it does, incremement it until the name isn't taken, and
  # concactanate the new value.

  while (exists($taken_names{"$type0$index"})) {
  $index++;
  }

  $taken_names{"$type0$index"} = 1;
  $type .= "0$index";

  # Finally, make the types hash contain a list with the current
  # label that is the same as the list the qcodes hash contained.
  $types{$type} = [ @{$scratch_hash{$_}} ];

 }


 # The last step is to give the new hash to the molecule we were
 # called with
 ${$molref}{types} = { %types };

 # Uncomment the following to display the contents of this hash
 # print "Displaying information in the types hash\n";
 # Look at the types information
 # %scratch_hash = %{$parent_molecule{types}};
 # print "Atoms with given types:\n";
 # foreach (keys(%scratch_hash)) {
 #  print "$_:  " . join(", ", @{$scratch_hash{$_}}) . "\n";
 # }

 # End assign_unique_names
 return;
 }
}


# The following function attaches an 'str' key to the given hash,
# which is the contents of the str file for that molecule.  It does
# not write out the file, it only attaches the data.  If you want the
# str file written out, it must be done manually.
sub attach_str_file(\%) {
 my $molref = shift;
 my %this_mol = %{$molref};
 my $str_file;

 $str_file .= "#\n";
 $str_file .= "# STR file automatically generated by:\n";
 $str_file .= "# $0\n";
 $str_file .= "# " . localtime() . "\n";
 $str_file .= "#\n";

 # With the preamble out of the way, get the good info into the file.
 $str_file .= scalar( @{$this_mol{coordinates}} ) . "\n";

 # Before we generate the coordinates, reverse the types hash so we
 # can easily insert the types.
 my (%typeshash);
 foreach (keys( %{ $this_mol{types} } ) ) {
 my (@atomlist) = @{$this_mol{types}{$_}};
 my $typelabel = $_;
 foreach (@atomlist) {
   $typeshash{$_} = $typelabel;
 }
 }

 # Now, generate the geometry lines of the str file.
 my $i;
 for ($i = 0; $i <= $#{$this_mol{coordinates}}; $i++ ) {
 $str_file .= $i . " ";
 $str_file .= $typeshash{$i}. " ";
 $str_file .= sprintf("%.8f ", $this_mol{coordinates}[$i][0]);
 $str_file .= sprintf("%.8f ", $this_mol{coordinates}[$i][1]);
 $str_file .= sprintf("%.8f ", $this_mol{coordinates}[$i][2]);
 $str_file .= sprintf("%.8f ", $this_mol{charges}[$i]);
 $str_file .= "\n";
 }

 # Beautiful.  The entire connectivity/charges lines have been
 # generated.  Now we generate the connectivity lines.

 # Instead of reversing the hash, we'll output the information
 # directly, but we'll also maintin a 'blacklist' of things that have
 # already been printed.  Before printing any bond, we will reference
 # that hash.  The key will be in the format <$atom1 $atom2>, with
 # $atom1 being the smaller of the two.  The values will simply be 1.
 my %blacklist;
 my %scratch_hash = %{$this_mol{connectivity}};
 my @bondlist;
 foreach ( sort {$a <=> $b} keys(%scratch_hash) ) {
 my $atom1 = $_;
 my @bondedtolist = @{$scratch_hash{$atom1}};
 foreach (sort {$a <=> $b} @bondedtolist) {
   my $atom2 = $_->[0];
   my $bond_order = $_->[1];
   if ($atom1 < $atom2) {
   unless ( exists($blacklist{"$atom1 $atom2"})) {
     push(@bondlist, "$atom1 $atom2 $bond_order");
     $blacklist{"$atom1 $atom2"} = 1;
   }
   } else {
```

```
    unless ( exists($blacklist{"$atom2 $atom1"}) ) {
      push(@bondlist, "$atom2 $atom1 $bond_order");
      $blacklist{"$atom2 $atom1"} = 1;
    }
    }
  }
  }

  $str_file .= scalar(@bondlist) . "\n";
  $str_file .= join("\n", @bondlist) . "\n";

  # As far as I can tell, that's all there is to the contents of
  # Matt's .str files.  If there's more information, it can be added
  # here later.
  ${$molref}{str} = $str_file;

  return;
}

# This will probably be the trickiest of the functions ... only so
# far, it turns out some of the other work is a bit trickier.  It must
# create an absorbed hydrogens molecule (the second hash), and must
# place 'tosister' values in both to map the values of which atoms
# came from where.  The atoms which have had hydrogens absorbed will
# have their name changed from C_XXX to C1XX or whatever to represent
# how many hydrogens were absorbed.  Both hashes will have a new
# 'tosister' field to map which atoms in each of the molecules
# correspond.

# Note: As of 4-22-02, the naming scheme has changed again.  Things
# that will have their hydrogens absorbed will have a slightly
# different name.  When we assigned unique names, the convention is to
# do like this C_0xxx, or whatever, so the names are changed to C_1xxx.

sub absorb_hydrogens(\%\%) {
  my $pmolref = shift;
  my $amolref = shift;
  my %this_parent = %{$pmolref};
  my %this_abs;

  my %ptoa_index_map;

  # Before we start any copying, we need to prepare some data for
  # later usage
  my %ptypesmap;  # Reversed types hash for parent molecule
  my %atypesmap;  # Reversed types hash for absorbed molecule
  foreach ( keys( %{$this_parent{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$this_parent{types}{$_}};
  foreach (@atomlist) {
    $ptypesmap{$_} = $type;
  }
  }

  # The first step is to copy all of the non-absorbable atoms (all
  # atoms but hydrogens which will be absorbed) into %this_abs, while
  # creating a hash that contains as keys, parent molecule atoms, and
  # as values, absorbed molecule atoms.  This hash can simply be
  # reversed for inclusion into the absorbed fragment.  In the first
  # pass, we only copy the fields that do not need to be mangled.

  # The keys that need to be copied into %this_abs are:
  #   charges:         (needs to be mangled)
  #   connectivity:    (needs to be mangled)
  #   coordinates:
  #   atom label:
  #   stereochemistry; (needs to be mangled)
  #   types:           (needs to be mangled)

  my $i;
  for ( $i = 0; $i <= $#{$this_parent{labels}}; $i++ ) {
  my $label = ${$this_parent{labels}}[$i];

  # Simply go to the next one if it's a hydrogen that will end up
  # being absorbed.
  my $max_bond_order = 0;
  if ( $label eq 'H' ) {
    my $next_atom = ${$this_parent{connectivity}{$i}}[0][0];
    my $this_label = $this_parent{labels}[$next_atom];
    # We only want to absorb hydrogens on carbons that are not
    # aromatic, the following test does that.
    if ($this_label eq 'C') {
    foreach ( @{$this_parent{connectivity}{$next_atom}} ) {
      my $bond_order = ${$_}[1];
      if ($bond_order > $max_bond_order) {
      $max_bond_order = $bond_order;
      }
    }
    unless ($max_bond_order > 1) {
      next;
    }
    }

  }

  # If we got past the last test section, we will be copying this
  # atom to %this_abs
  my $patom_index = $i;
  my $aatom_index = defined($this_abs{coordinates}) ?
    scalar(@{$this_abs{coordinates}}) : 0;

  # Let the copying commence
  $ptoa_index_map{$patom_index} = $aatom_index;

  # I don't want the coordinates of the absorbed fragment to simply
  # point to the same place that the parent atom coordinates are, I
  # want my own copy, thus the funky [ @{ } ] expression in the
  # following assignment

  $this_abs{labels}[$aatom_index] = $label;

  $this_abs{coordinates}[$aatom_index] =
    [ @{$this_parent{coordinates}[$patom_index]} ];
  }

  # The coordinates are now into %this_abs.  We will now go through
  # each of the atoms on %this_abs (via the reversed hash), and add in
  # all of the mangled values;
  my %atop_index_map;
  foreach (keys(%ptoa_index_map)) {
  $atop_index_map{$ptoa_index_map{$_}} = $_;
  }

  # Note that if an atom exists on the %ptoa_index_map, then it will
  # be an unabsorbed atom.  We can use this to determine if any of the
  # atoms connected to the atom we're at in the %atop_index_map should
  # be absorbed (as far as adding in charges, or whatnot);

  foreach (keys(%atop_index_map)) {
  my $aatom = $_;
  my $patom = $atop_index_map{$_};

  # First, let's get the charges right;
  my $charge = $this_parent{charges}[$patom];
  my $absorbed_H_count = 0;

  # Loop through the atoms that this atom 'should be' connected to.
  foreach ( @{$this_parent{connectivity}{$patom}} ) {
    my $connected_atom = $_->[0];
    # If this atom doesn't exist in the %ptoa_index_map, we need to
    # absorb it's charge onto the fragment atom and increment
    # $absorbed_H_count, so we can re-do it's type.  We'll copy the
    # connectivity in another pass.
    unless (exists($ptoa_index_map{$connected_atom})) {
    $charge += $this_parent{charges}[$connected_atom];
    $absorbed_H_count++;
    }
  }

  # Put the charge on the absorbed fragment
  ${$this_abs{charges}}[$aatom] = $charge;

  # Put the proper type on the absorbed fragment (reversed) types map.
  if ($absorbed_H_count == 0) {
    # Copy the type verbatim
    $atypesmap{$aatom} = $ptypesmap{$patom};
  } else {
    # Mangle the label a bit, then copy it.
    my $ptype = $ptypesmap{$patom};

    # The emacs CPerl mode was giving my trouble with coloring, I
    # added the next line to keep it happy.  Turns out it eventually
    # got unhappy with that as well, we'll simply do the code
    # differently.

    # Remember, we only mangle the label if it's a carbon.
    if ($ptype =~ /^C_/ ) {
    my @oldlabel = split('', $ptype);
    $oldlabel[2] = $absorbed_H_count;
    $ptype = join ('', @oldlabel);
    }

    $atypesmap{$aatom} = $ptype;
  }
  }

  # And put the reversed map into %this_abs
  foreach (keys %atypesmap ) {
  push(@{$this_abs{types}{$atypesmap{$_}}}, $_ );
  }

  # The types, charges, and coordinates are done.  Map any
  # stereochemistry that may be needed.
  foreach (@{$this_parent{stereochemistry}}) {
  my $patomindex = $_->[0];
  my $patomval = $_->[1];

  if (exists($ptoa_index_map{$patomindex})) {
    push(@{$this_abs{stereochemistry}},
       [ $ptoa_index_map{$patomindex}, $patomval ]);
  }
  }

  # Everything is done but the connectivity.  This will be done in a
  # single pass.  For every atom that exists in the %ptoa_atom_index,
  # we transcribe the connectivity to %this_abs.
  foreach (keys(%atop_index_map)) {
  my $aatom = $_;
  my $patom = $atop_index_map{$_};

  # Loop through the atoms that this atom 'should be' connected to.
  foreach ( @{$this_parent{connectivity}{$patom}} ) {
    my $connected_atom = $_->[0];
    my $bond_order = $_->[1];
    # If this atom exists in the %ptoa_index_map, we need to
    # copy any relavent connectivity
    if (exists($ptoa_index_map{$connected_atom})) {
    $aconnected_atom = $ptoa_index_map{$connected_atom};

    push(@{$this_abs{connectivity}{$aatom}},
       [ $aconnected_atom, $bond_order ]);
    }
  }
  }

  # That's it, we now have a proud new fragment, with all of the
  # proper values absorbed.  Set the original second hash to our newly
  # built absorbed fragment, and connect the tranlation hashes to each
  # of the two we were called with.
  $this_abs{tosister} = { %atop_index_map };
  %{$amolref} = %this_abs;

  ${$pmolref}{tosister} = { %ptoa_index_map };

  return;
}
```

```
# Like the previous function, this function does everything it does,
# but actually absorb the hydrogens.  Duplicating the information is
# definately redundant, but this is kind of a last minute 'feature' addition.
sub dont_absorb_hydrogens(\%\%) {
  my $pmolref = shift;
  my $amolref = shift;
  my %this_parent = %{$pmolref};
  my %this_abs;

  my %ptoa_index_map;

  # Before we start any copying, we need to prepare some data for
  # later usage
  my %ptypesmap;  # Reversed types hash for parent molecule
  my %atypesmap;  # Reversed types hash for absorbed molecule
  foreach ( keys( %{$this_parent{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$this_parent{types}{$_}};
  foreach (@atomlist) {
    $ptypesmap{$_} = $type;
  }
  }

  # The first step is to copy all of the atoms into %this_abs, while
  # creating a hash that contains as keys, parent molecule atoms, and
  # as values, absorbed molecule atoms.  This hash can simply be
  # reversed for inclusion into the absorbed fragment.  In the first
  # pass, we only copy the fields that do not need to be mangled.

  # The keys that need to be copied into %this_abs are:
  #   charges:         (needs to be mangled)
  #   connectivity:    (needs to be mangled)
  #   coordinates:
  #   atom label:
  #   stereochemistry; (needs to be mangled)
  #   types:           (needs to be mangled)

  my $i;
  for ( $i = 0; $i <= $#{$this_parent{labels}}; $i++ ) {
  my $label = ${$this_parent{labels}}[$i];

  # We will be copying this atom to %this_abs
  my $patom_index = $i;
  my $aatom_index = defined($this_abs{coordinates}) ?
    scalar(@{$this_abs{coordinates}}) : 0;

  # Let the copying commence
  $ptoa_index_map{$patom_index} = $aatom_index;

  # I don't want the coordinates of the absorbed fragment to simply
  # point to the same place that the parent atom coordinates are, I
  # want my own copy, thus the funky [ @{ } ] expression in the
  # following assignment

  $this_abs{labels}[$aatom_index] = $label;

  $this_abs{coordinates}[$aatom_index] =
    [ @{$this_parent{coordinates}[$patom_index]} ];

  }

  # The coordinates are now into %this_abs.  We will now go through
  # each of the atoms on %this_abs (via the reversed hash), and add in
  # all of the mangled values;
  my %atop_index_map;
  foreach (keys(%ptoa_index_map)) {
  $atop_index_map{$ptoa_index_map{$_}} = $_;
  }

  # Note that if an atom exists on the %ptoa_index_map, then it will
  # be an unabsorbed atom.  We can use this to determine if any of the
  # atoms connected to the atom we're at in the %atop_index_map should
  # be absorbed (as far as adding in charges, or whatnot);

  foreach (keys(%atop_index_map)) {
  my $aatom = $_;
  my $patom = $atop_index_map{$_};

  # First, let's get the charges right;
  my $charge = $this_parent{charges}[$patom];
  my $absorbed_H_count = 0;

  # Loop through the atoms that this atom 'should be' connected to.
  foreach ( @{$this_parent{connectivity}{$patom}} ) {
    my $connected_atom = $_->[0];
    # If this atom doesn't exist in the %ptoa_index_map, we need to
    # absorb it's charge onto the fragment atom and increment
    # $absorbed_H_count, so we can re-do it's type.  We'll copy the
    # connectivity in another pass.
    unless (exists($ptoa_index_map{$connected_atom})) {
    $charge += $this_parent{charges}[$connected_atom];
    $absorbed_H_count++;
    }
  }

  # Put the charge on the absorbed fragment
  ${$this_abs{charges}}[$aatom] = $charge;

  # Put the proper type on the absorbed fragment (reversed) types map.
  if ($absorbed_H_count == 0) {
    # Copy the type verbatim
    $atypesmap{$aatom} = $ptypesmap{$patom};
  } else {
    # Mangle the label a bit, then copy it.
    my $ptype = $ptypesmap{$patom};

    # The emacs CPerl mode was giving my trouble with coloring, I
    # added the next line to keep it happy.  Turns out it eventually
    # got unhappy with that as well, we'll simply do the code
    # differently.

    # Remember, we only mangle the label if it's a carbon.
    if ($ptype =~ /^C_/ ) {
    my @oldlabel = split('', $ptype);
    $oldlabel[2] = $absorbed_H_count;
    $ptype = join ('', @oldlabel);
    }

    $atypesmap{$aatom} = $ptype;
  }
  }

  # And put the reversed map into %this_abs
  foreach (keys %atypesmap ) {
  push(@{$this_abs{types}{$atypesmap{$_}}}, $_ );
  }

  # The types, charges, and coordinates are done.  Map any
  # stereochemistry that may be needed.
  foreach (@{$this_parent{stereochemistry}}) {
  my $patomindex = $_->[0];
  my $patomval = $_->[1];

  if (exists($ptoa_index_map{$patomindex})) {
    push(@{$this_abs{stereochemistry}},
       [ $ptoa_index_map{$patomindex}, $patomval ]);
  }
  }

  # Everything is done but the connectivity.  This will be done in a
  # single pass.  For every atom that exists in the %ptoa_atom_index,
  # we transcribe the connectivity to %this_abs.
  foreach (keys(%atop_index_map)) {
  my $aatom = $_;
  my $patom = $atop_index_map{$_};

  # Loop through the atoms that this atom 'should be' connected to.
  foreach ( @{$this_parent{connectivity}{$patom}} ) {
    my $connected_atom = $_->[0];
    my $bond_order = $_->[1];
    # If this atom exists in the %ptoa_index_map, we need to
    # copy any relavent connectivity
    if (exists($ptoa_index_map{$connected_atom})) {
    $aconnected_atom = $ptoa_index_map{$connected_atom};

    push(@{$this_abs{connectivity}{$aatom}},
       [ $aconnected_atom, $bond_order ]);
    }
  }
  }

  # That's it, we now have a proud new fragment, with all of the
  # proper values absorbed.  Set the original second hash to our newly
  # built absorbed fragment, and connect the tranlation hashes to each
  # of the two we were called with.
  $this_abs{tosister} = { %atop_index_map };
  %{$amolref} = %this_abs;

  ${$pmolref}{tosister} = { %ptoa_index_map };

  return;
}

# The following fuction is purely diagnostic, but it allows us to
# check our molecules for correctness in coordinates and connectivity;
sub make_com_file(\%) {
  my $molref = shift;
  my %this_mol = %{$molref};

  print "#p opt ram1 geom=connectivity\n\n";
  print "No comment\n\n";
  print "0 1\n";

  # Print out the coordinates (the simple part)
  my $i;
  for ($i = 0; $i <= $#{$this_mol{labels}}; $i++) {
  print "${$this_mol{labels}}[$i],";
  printf "%.8f,", ${$this_mol{coordinates}}[$i][0];
  printf "%.8f,", ${$this_mol{coordinates}}[$i][1];
  printf "%.8f\n", ${$this_mol{coordinates}}[$i][2];
  }

  print "\n";

  # Print out the connectivity the way gaussian likes it (the hard
  # part).  Once again, we'll be using the idea of a blacklist, of
  # atoms already printed
  my %blacklist;
  my %scratch_hash = %{$this_mol{connectivity}};
  my @bondlist;
  foreach ( sort {$a <=> $b} keys(%scratch_hash) ) {
  my $atom1 = $_;
  my $thisline = ($atom1 + 1) . "  ";
  my @bondedtolist = @{$scratch_hash{$atom1}};
  foreach (sort {$a <=> $b} @bondedtolist) {
    my $atom2 = $_->[0];
    my $bond_order = $_->[1];
    if ($atom1 < $atom2) {
    unless ( exists($blacklist{"$atom1 $atom2"})) {
      $thisline .= ($atom2 + 1) . " ";
      $thisline .= sprintf("%.1f  ", $bond_order );
      $blacklist{"$atom1 $atom2"} = 1;
    }
    } else {
    unless ( exists($blacklist{"$atom2 $atom1"}) ) {
      $thisline .= ($atom1 + 1) . " ";
      $thisline .= sprintf("%.1f  ", $bond_order );
      $blacklist{"$atom2 $atom1"} = 1;
    }
    }
  }
  push(@bondlist, $thisline);
  }

  print join("\n", @bondlist) . "\n";
```

```
  return;
}

# Aside from output, this is the last (and busiest) bit of work that
# this program must do.  It attaches an 'mff' field to the given hash,
# so a completed force field (with 0 values for all the torsions) can
# be output.  It will use several sub-functions to assign various
# values (generic parameters).  They will be declared within the
# function, since only this function will use them.  They may later be
# exported for use by a more generic perl module.  Note that each
# function will be declared directly before it's first use.  This is
# strange, but this program is also a prototype :-).  In retrospect, I
# should have simply made an 'assign_amber_type' function for the
# typing, but I won't do that now.  Code for such a function could be
# robbed from the individual funtions that follow.
sub attach_mff_file(\%) {

  my $molref = shift;
  my %this_mol = %{$molref};
  my $mff;

  my $time = localtime();
  $mff = <<PREAMBLE
#
# Force field created by $0 on
# $time
#
# All bond stretches and angle bending taken from Drieding 2.
#
# vdW parameters are taken from a number of sources, most notably from
# Jorgensen (OPLS).
#
# United atom vdW parameters for CH2 and CH3 from J.I. Siepmann, S.
# Karaborni, and B. Smit, Nature 365, 330 (1993).
#
# UA vdW parameters for CH1 from W.L. Jorgensen, J.D. Madura, and
# C.J. Swenson, J. Am. Chem. Soc. 106, 6638 (1994).
#
# vdW parameters for phenyl C and H from W.L. Jorgensen et al.,
# J. Comput. Chem. 14, 206 (1993). Note: the H vdW parameters from
# this paper are used for all explicit H's (e.g. in alcohols).
#
# vdW parameters for carbonyl C and O from J.M. Briggs et al., J. Phys.
# Chem. 95, 3315 (1991).
#
PREAMBLE
;

  # With the preamble out of the way, we can start adding the other
  # necessary sections;
  sub print_masses(\%);

  # Get the masses section of the mff file.
  $mff .= print_masses(%this_mol);

  # Get the vdw section of the mff file.
  sub print_vdw(\%);
  $mff .= print_vdw(%this_mol);

  # Get the stretch section of the mff file.
  sub print_stretch(\%);
  $mff .= print_stretch(%this_mol);

  # Get the bend section of the mff file;
  sub print_bend(\%);
  $mff .= print_bend(%this_mol);

  # Get generic inversion parameters
  sub print_generic_inversion(\%);
  $mff .= print_generic_inversion(%this_mol);

  # Get generic torsion parameters
  sub print_generic_torsions(\%);
  $mff .= print_generic_torsions(%this_mol);

  # Get generic torsion parameters
  sub print_other_blank_torsions(\%);
  $mff .= print_other_blank_torsions(%this_mol);

  # That's it, a shiny new mff file is attached to the molecule.

  ${$molref}{mff} = $mff;
  return;
}

# This function is responsible for printing the mass section of an mff
# file, which also includes the absorbed types.  It is enclosed in a
# begin section so the masses can be initialized only once.  The
# values were taken from:
# http://www.slvhs.slv.k12.ca.us/~pboomer/chemlectures/textass2/table8-3.html
BEGIN {

  my %masses = qw/
Ac 227
Al 26.98
Am 243
Sb 121.76
Ar 39.944
As 74.91
At 210
Ba 137.36
Bk 247
Be 9.013
Bi 209
B 10.82
Br 79.916
Cd 112.41
Ca 40.08
Cf 251
C 12.011
Ce 140.13
Cs 132.91
Cl 35.457
Cr 52.01
Co 58.94
Cu 63.54
Cm 247
Dy 162.51
Fm 253
F 19.00
Fr 223
Gd 157.26
Ga 69.72
Ge 72.60
Au 197.0
Hf 178.50
He 4.003
Ho 164.94
H 1.0080
In 114.82
I 126.91
Ir 192.2
Fe 55.85
Kr 83.80
La 138.92
Pb 207.21
Li 6.940
Lu 174.99
Mg 24.32
Mn 54.94
Md 256
Hg 200.61
Mo 95.95
Nd 144.27
Ne 20.183
Np 237
Ni 58.71
Nb 92.91
N 14.008
No 253
Os 190.2
O 16.0000
Pd 106.4
P 30.975
Pt 195.09
Pu 244
Po 210
K 39.100
Pr 140.92
Pm 145
Pa 213
Ra 226
Re 186.22
Rh 102.91
Rb 85.48
Ru 101.10
Sm 150.35
Sc 44.96
Se 78.96
Si 28.09
Ag 107.88
Na 22.991
Sr 87.63
S 32.066
Ta 180.95
Tc 98
Te 127.61
Tb 158.93
Tl 204.39
Th 232
Tm 168.94
Sn 118.70
Ti 47.90
U 238
V 50.95
Xe 131.30
Yb 173.04
Y 88.92
Zn 65.38
Zr 91.22
/;

  sub print_masses (\%) {
  my $molref = shift;
  my %mol = %{$molref};
  my $ret;  # For return string

  $ret .= "#\n";
  $ret .= "# mass\n";
  foreach (keys %{$mol{types}} ) {
    my $fulltype = $_;
    my $realtype;

    # Play with realtype (from $fulltype) to get a real atomic label
    # out of it.
    $fulltype =~ /([[:upper:]][[:lower:]]*_|C[123])/;
    $realtype = $1;

    # If we have a trailing underscore, strip it.
    if ($realtype =~ /([^_]+)_/) {
    $realtype = $1;
    }

    # If our $fulltype isn't an absorbed type of carbon, simply
    # print the real masses, otherwise, add in the absorbed
    # hydrogen masses.
    if ($fulltype =~ /^C_([123])/ ) {
    $ret .= "$fulltype ";
    my $realmass = $masses{C} + $1 * $masses{H};
    $ret .= "$masses{C} $realmass\n";
    } elsif ( $realtype eq 'H' ) {
        $ret .= "$fulltype $masses{H} $masses{C}\n";
    } else {
    my $thistype;
    $ret .= "$fulltype $masses{$realtype} $masses{$realtype}\n";
    }
  }
```

```
  $ret .= "# end\n";
  $ret .= "#\n";
  return $ret;
  }

}

# This function returns our 'current' van der waals values for
# diagonal interactions of the atom types.  Once again, we'll have
# $fulltypes and $realtypes for the atoms, and we'll return the values
# for the interactions among $fulltypes.  Unfortunately, this is a
# tricky function with many test cases to categorize the atom in
# question.
sub print_vdw(\%) {
  my $molref = shift;
  my %mol = %{$molref};
  my $ret;            # For return string

  $ret .= "#\n";
  $ret .= "# Parameters for nitro oxygen and nitrogen taken from\n" .
        "# J. Comp. Chem, Vol. 22, No. 13, 1340-1352, (2001)\n" .
      "#\n" .
      "# Parameters for amine nitrogenstaken from\n" .
      "# JACS, 1999, 121, 4827-4836\n" .
      "#\n";
  $ret .= "# vdw\n";

 TYPES:  foreach (keys %{$mol{types}} ) {
  my $fulltype = $_;
  my $realtype;

  # Play with realtype (from $fulltype) to get a real atomic label
  # out of it.
  $fulltype =~ /([[:upper:]][[:lower:]]*_\d)/;
  $realtype = $1;

  # If our $realtype is defined, we have a carbon with absorbed
  # hydrogens, and we return the appropriate parameters.
  if ($realtype eq 'C_1') {
      $ret .= "$fulltype $fulltype 3.850 0.08000 0.0 0.0 0.0\n";
      next;
  } elsif ($realtype eq 'C_2') {
      $ret .= "$fulltype $fulltype 3.930 0.09334 0.0 0.0 0.0\n";
      next;
  } elsif ($realtype eq 'C_3') {
      $ret .= "$fulltype $fulltype 3.930 0.08000 0.0 0.0 0.0\n";
      next;
  } else {
      # It's not an absorbed carbon, so simply chop off the last
      # character, so it matches the way we request.
      $realtype =~ /(.*)\d/;
      $realtype = $1;
  }

  # If our $realtype was defined, then we inserted the parameters
  # to the absorbed Carbon type already.  Otherwise, we need to
  # figure out what type of atom we're playing with, from most
  # restrictive to least restrictive.  This involves exploring what
  # is bonded to the current atom, etc.

  # If our atom is an oxygen, we have 3 choices.  It's either an
  # sp3 hybridized oxygen, an sp2 hybridized oxygen, or an
  # resonant oxygen.

  if ($fulltype =~ /^O_/ ) {
    # For this type, we need to choose one of the three.  This
    # requires knowing who's connected to it, and what the
    # properties of those atoms are.  We'll arbitrarily pick the
    # first atom, since if several atoms share this type, they
    # should have the same environment.
    my $this_atom = ${$mol{types}{$fulltype}}[0];
    my @neighbors = @{$mol{connectivity}{$this_atom}};
    my $neighbor_count = scalar(@neighbors);

    if ($neighbors[0][1] == 2) {
    # This is a ketone oxygen.
    $ret .= "$fulltype $fulltype 2.960 0.210 0.0 0.0 0.0\n";
    next;
    }

    # It might also be an oxygen on a nitro, let's check that.
    if ($neighbors[0][1] == 1.5 and
      $mol{labels}[$neighbors[0][0]] eq 'N') {
    $ret .= "$fulltype $fulltype 2.960 0.170 0.0 0.0 0.0\n";
    next;
    }

    # If the oxygen is connected to any atoms who have max bond
    # orders greater than 1, it is resonant.

    my $max_bond_order = 0;
    foreach (@neighbors) {
    my @this_neighbors_neighbors = @{$mol{connectivity}{$_->[0]}};
    foreach (@this_neighbors_neighbors) {
      if ($_->[1] > $max_bond_order ) {
      $max_bond_order = $_->[1];
      }
    }
    }

    if ($max_bond_order > 1) {
    # It must be some other sort of a resonant oxygen.
    $ret .= "$fulltype $fulltype 3.550 0.070 0.0 0.0 0.0\n";
    next;
    }

    # If we've gotten this far, it must be a simple sp3 oxygen.
    # Use the generic oxygen parameters.
    $ret .= "$fulltype $fulltype 3.000 0.170 0.0 0.0 0.0\n";
    next;

  }

  # Oxygen is done, let's handle nitrogen.  The two cases we need
  # to worry about are nitro oxygens, and amine oxygens.  More
  # specialized cases can be added later.

  if ($fulltype =~ /^N_/ ) {

    my $this_atom = ${$mol{types}{$fulltype}}[0];
    my @neighbors = @{$mol{connectivity}{$this_atom}};
    my $neighbor_count = scalar(@neighbors);
    my $O_neighbor_count = 0;
    my $max_bond_order = 0;

    foreach (@neighbors) {
    if (${$mol{labels}}[$_->[0]] eq 'O'  ) {
      $O_neighbor_count++;
    }
    if ($_->[1] > $max_bond_order ) {
      $max_bond_order = $_->[1];
    }
    }

    if ($O_neighbor_count == 2 ) {
    # This must be a nitro group nitrogen.
    $ret .= "$fulltype $fulltype 2.960 0.170 0.0 0.0 0.0\n";
    next;
    }

    # If it's not a nitro, it must be some kind of aliphatic
    # nitrogen.  Note that the second referenced paper in the header
    # here contains parameters for many kinds of nitrogen, but they
    # are identical as far as the VDW parameters go.
    $ret .= "$fulltype $fulltype 3.300 0.170 0.0 0.0 0.0\n";
    next;

  }

  # H's are tremendously simple, we'll do them next.
  if ($fulltype =~ /^H_/ ) {
    $ret .= "$fulltype $fulltype 2.420 0.030 0.0 0.0 0.0\n";
    next;
  }

  # And finally, get the type for carbons.  Note that if any atom
  # actually makes it past these tests, it is an error, and we
  # will die immediately.
  if ($fulltype =~ /^C_/ ) {

    my $this_atom = ${$mol{types}{$fulltype}}[0];
    my @neighbors = @{$mol{connectivity}{$this_atom}};
    my $neighbor_count = scalar(@neighbors);
    my $O_neighbor_count = 0;
    my $max_bond_order = 0;

    # If the carbon we're looking at has 3 carbon neighbors, all
    # with maximum bond orders of 1.5, it is a bridgehead resonant
    # carbon.
    my $resonant_carbon_neighbor_count = 0;
    foreach (@neighbors) {
    my @this_neighbors_neighbors = @{$mol{connectivity}{$_->[0]}};
    my $is_carbon = 0;
    my $nmax_bond_order = 0;

    if (${$mol{labels}}[$_->[0]] eq 'C' ) {
      $is_carbon = 1;
    }

    foreach (@this_neighbors_neighbors) {
      if ($_->[1] > $nmax_bond_order ) {
      $nmax_bond_order = $_->[1];
      }
    }

    if ( $is_carbon and $nmax_bond_order == 1.5 ) {
      $resonant_carbon_neighbor_count++;
    }
    $is_carbon = 0;
    $nmax_bond_order = 0;
    }

    # Now we can test to see if it is a bridgehead carbon.
    if ( $resonant_carbon_neighbor_count == 3) {
    $ret .= "$fulltype $fulltype 3.550 0.070 0.0 0.0 0.0\n";
    next;
    } elsif ($resonant_carbon_neighbor_count == 2) {
    $ret .= "$fulltype $fulltype 3.550 0.070 0.0 0.0 0.0\n";
    next;
    }

    # Ok, it's not a normal aromatic carbon, or a bridgehead
    # carbon, see if it's a carbonyl carbon.
    foreach (@neighbors) {
    my $this_bond_order = $_->[1];
    if (${$mol{labels}}[$_->[0]] eq 'O' and
      $this_bond_order == 2 ) {
      # We can definately call this a carbonyl carbon
      $ret .= "$fulltype $fulltype 3.750 0.105 0.0 0.0 0.0\n";
      next TYPES;
    }
    }

    # What is left is normal aliphatic carbons.  Since we don't have
    # any parameters for them, we'll simply die if we run into one.
    # Since we're now using unabsorbed models sometimes, we will not
    # die outright, we'll check first.
  unless ($explicit_h) {
    die "Unhandled carbon type $fulltype at atom (1 base) " .
    (${$mol{types}{$fulltype}}[0] + 1);
  } else {
      $ret .= "$fulltype $fulltype 3.500 0.066 0.0 0.0 0.0\n";
    next TYPES;
  }

  }

  die "Unknown atom type $fulltype encountered at atom number " .
```

```
    "(1 base) " . (${$mol{types}{$fulltype}}[0] + 1) . "";

  }

  $ret .= "# end\n";
  $ret .= "#\n";

  return $ret;
}

# This function creates the stretch part of matt's force field file.
sub print_stretch(\%) {
  my $molref = shift;
  my %mol = %{$molref};
  my $ret;  # For return string

  $ret .= "# Bond lenghts and bond angles are taken from the current\n" .
        "# structure (which was optimized in the qdb).  Force constants\n" .
      "# are generic Dreiding 2 values\n";
  $ret .= "#\n";
  $ret .= "# stretch\n";

  my %ptypesmap;
  # Firstly, we reverse the types hash.
  foreach ( keys( %{$mol{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$mol{types}{$_}};
  foreach (@atomlist) {
    $ptypesmap{$_} = $type;
  }
  }


  # Get one copy of each bond.  Note that if there are bonds with the
  # same atom types, but different bond lenghts, they must be
  # averaged.  We will be reprocessing the initial list into a hash
  # that will be much easier to manipulate and find identical members of.
  my %blacklist;
  my %scratch_hash = %{$mol{connectivity}};
  my @bondlist;
  foreach ( sort {$a <=> $b} keys(%scratch_hash) ) {
  my $atom1 = $_;
  my @bondedtolist = @{$scratch_hash{$atom1}};
  foreach (sort {$a <=> $b} @bondedtolist) {
    my $atom2 = $_->[0];
    my $bond_order = $_->[1];
    if ($atom1 < $atom2) {
    unless ( exists($blacklist{"$atom1 $atom2"})) {
      push(@bondlist, "$atom1 $atom2 $bond_order");
      $blacklist{"$atom1 $atom2"} = 1;
    }
    } else {
    unless ( exists($blacklist{"$atom2 $atom1"}) ) {
      push(@bondlist, "$atom2 $atom1 $bond_order");
      $blacklist{"$atom2 $atom1"} = 1;
    }
    }
  }
  }

  my %bondhash;
  my $i = 0;
  foreach (@bondlist) {
  my %tmphash;
  my $spring_constant;
  my ($atom1, $atom2, $bond_order) = split(" ", $_);
  my @types = ($ptypesmap{$atom1}, $ptypesmap{$atom2});
  @types = sort @types;

  if ($bond_order == 1) {
    $spring_constant = 700;
  } elsif ($bond_order == 1.5) {
    $spring_constant = 1050;
  } elsif ($bond_order == 2) {
    $spring_constant = 1400
  } else {
    die "Constant for bond order $bond_order unknown\n";
  }

  my @atom1_coordinates = @{${$mol{coordinates}}[$atom1]};
  my @atom2_coordinates = @{${$mol{coordinates}}[$atom2]};

  my @difference = v_sub(@atom1_coordinates, @atom2_coordinates);

  my $length = v_scalar_len(@difference);

  $tmphash{atom1} = $atom1;
  $tmphash{atom2} = $atom1;
  $tmphash{spring} = $spring_constant;
  $tmphash{type1} = $types[0];
  $tmphash{type2} = $types[1];
  $tmphash{length} = $length;

  $bondhash{$i} = { %tmphash };
  $i++;
  }

  # Now, we can finally look for duplicates.  For each of the
  # duplicates, we'll average the values, modify one of them, and
  # delete the rest.  These will go into 'outhash' with the key being
  # the output line (as it's being built), and the value being a space
  # separated list, with the first value being the number of values
  # already averaged in, and the second number being the current bond
  # lenghth.
  my %outhash;
  foreach (sort {$a <=> $b} keys(%bondhash) ) {
  my %thishash = %{$bondhash{$_}};
  my $line = "$thishash{type1} $thishash{type2} $thishash{spring}";

  if (exists($outhash{$line})) {
    my ($count, $length) = split(" ", $outhash{$line});
    $length = ($length * $count + $thishash{length}) / ($count + 1);
    $count++;
    $outhash{$line} = "$count $length";
  } else {
    $outhash{$line} = "1 $thishash{length}";
  }
  }

  # Now we're ready to create the output lines
  foreach (sort keys (%outhash )) {
  my ($count, $length) = split(" ", $outhash{$_});
  $ret .= "$_ $length\n";
  }

  $ret .= "# end\n";
  $ret .= "#\n";
  return $ret;
}

# This function creates the stretch part of matt's force field file.
sub print_bend(\%)  {
  my $molref = shift;
  my %mol = %{$molref};
  my $ret;  # For return string
  my %angles;

  $ret .= "#\n";
  $ret .= "# bend\n";

  my %ptypesmap;
  # Firstly, we reverse the types hash.
  foreach ( keys( %{$mol{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$mol{types}{$_}};
  foreach (@atomlist) {
    $ptypesmap{$_} = $type;
  }
  }

  # Go through the types hash, recording all atoms that are connected
  # to 2 other atoms, and discarding others.  Each member of angles
  # will have as it's key the atom number of the central atom, and as
  # it's value, a list of references to hashes which will themselves
  # have the values of the two connected atoms, their types, and the
  # angle.  For this pass, however, the values will just be set to 1.
  foreach (keys %ptypesmap) {
  my @connectivity = @{$mol{connectivity}{$_}};
  if (scalar(@connectivity) >= 2 ) {
    $angles{$_} = 1;
  }
  }

  # Now, for each of the atoms in $angles, we need to find each of the
  # possible angles, and record the relevant values, as outlines in
  # the last block of comments.
  foreach (keys %angles ) {
  my $central_atom = $_;
  $angles{$central_atom} = undef;
  my @central_coordinates = @{${$mol{coordinates}}[$_]};
  my @connectivity = @{$mol{connectivity}{$_}};
  my ($i, $j);
  for ($i = 0; $i <= $#connectivity - 1; $i++) {
    for ($j = $i + 1; $j <= $#connectivity; $j++) {
    # Here we are, record all of the relevant data.
    my %tmp;

    my @atom1_coordinates = @{${$mol{coordinates}}[$connectivity[$i][0]]};
    my @atom2_coordinates = @{${$mol{coordinates}}[$connectivity[$j][0]]};
    my @vec1 = v_sub(@atom1_coordinates, @central_coordinates);
    my @vec2 = v_sub(@atom2_coordinates, @central_coordinates);
    my $angle = acos(v_dot_prod(@vec1, @vec2) /
              (v_scalar_len(@vec1) * v_scalar_len(@vec2))
             ) * 180 / $PI;

    # We've got everything we need, shove it into the hash at this
    # location.
    $tmp{type1} = $ptypesmap{$connectivity[$i][0]};
    $tmp{type2} = $ptypesmap{$connectivity[$j][0]};
    $tmp{angle} = $angle;

    push(@{$angles{$central_atom}}, { %tmp });
    }
  }
  }

  # Now, much as we did for the bond stretching section, we look for
  # duplicates and average the values.
  my %outhash;
  foreach (sort {$a <=> $b} keys(%angles) ) {
  my @thislist = @{$angles{$_}};
  my $central_atom = $_;

  foreach (@thislist) {
    my %thishash = %{$_};
    # Sort the two ends of the angle, so we always look the same
    my @types = sort ($thishash{type1}, $thishash{type2} );
    my $line = "$types[0] $ptypesmap{$central_atom} $types[1] 100";

    if (exists($outhash{$line})) {
    my ($count, $angle) = split(" ", $outhash{$line});
    $angle = ($angle * $count + $thishash{angle}) / ($count + 1);
    $count++;
    $outhash{$line} = "$count $angle";
    } else {
    $outhash{$line} = "1 $thishash{angle}";
    }

  }

  }

  foreach (sort keys (%outhash )) {
  my ($count, $angle) = split(" ", $outhash{$_});
  $ret .= "$_ $angle\n";
  }

  $ret .= "# end\n";
```

```
  $ret .= "#\n";

  return $ret;
}

# Output a generic inversion section.  This is in fact very very
# simple, since we only have two kinds of inversions.  We have one
# generic inversion for sp3 carbons, and another type for everything
# else (that has 3 things connected to it)
sub print_generic_inversion(\%)  {
  my $molref = shift;
  my %mol = %{$molref};
  my $ret;  # For return string

  $ret .= "#\n";
  $ret .= "# inversion\n";

  # Simply find all atoms with three things around them, and output
  # the proper inversion for them.
  foreach (keys( %{$mol{types}} ) ) {
  # For each of these, we'll only look at the first atom in the
  # given list, since others should be suitably similar.
  my $this_atom = $mol{types}{$_}[0];
  my $this_type = $_;

  my @connectivity = @{$mol{connectivity}{$this_atom} };

  if (scalar(@connectivity ) == 3 ) {
    # This is an invertable atom, we need to provide generic
    # inversion parameters for it.  First, we'll find out if it's an
    # sp3 carbon.
    if ( $this_type =~ /^C_1/ ) {
    $ret .= "$this_type X X X 40 54.736 " .
            "# Generic sp3 inversion parameters\n";
    next;
    }

    # Put more test here later if we want to.
    $ret .= "$this_type X X X 40 0 " .
          "# Generic (any uncategorized type) parameters\n";
  }
  }

  $ret .= "# end\n";
  $ret .= "#\n";

  return $ret;
}

# Output a generic torsion section.  What we need to do is get a list
# of all pairs of aromatic carbons and provide generic (high energy)
# torsions for them.
sub print_generic_torsions(\%)  {
  my $molref = shift;
  my %mol = %{$molref};
  my $ret;  # For return string
  my %outhash;

  $ret .= "#\n";
  $ret .= "# torsion\n";

  # We'll need a reversed version of the types hash to work with.
  my %ptypesmap;
  foreach ( keys( %{$mol{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$mol{types}{$_}};
  foreach (@atomlist) {
    $ptypesmap{$_} = $type;
  }
  }

  # We're only generating generic torsions for pairs of aromatic
  # carbons.  A carbon is aromatic if it has three neighbors, and at
  # least two bonds that are of bond order 1.5.
  foreach (keys( %{$mol{types}} ) ) {
  # For each of these, we'll only look at the first atom in the
  # given list, since others should be suitably similar.
  my $this_atom = $mol{types}{$_}[0];
  my $this_type = $_;
  my $resonant_bond_count;

  # If it's not a carbon, just skip to the next one.
  unless ($this_type =~ /^C/ ) {
    next;
  }

  my @connectivity = @{$mol{connectivity}{$this_atom} };

  $resonant_bond_count = 0;
  if (scalar(@connectivity ) == 3 ) {
    # This atom may be aromatic, let's look through the connectivity
    # to find out.
    foreach (@connectivity) {
    my $bond_order = $_->[1];
    my $other_type = $ptypesmap{$_->[0]};

    if ($bond_order == 1.5 and $other_type =~ /^C/) {
      $resonant_bond_count++;
    }
    }
  }

  if ($resonant_bond_count >= 2) {
    # If we've got an aromatic carbon, and neighbors, we simply add
    # the line to our outhash.  The key will be the entire output
    # line, and the value will be simply 1.  Once again, we sort the
    # 2 labels so we don't get duplicate lines.
    my $label1 = $this_type;

    foreach (@connectivity) {
    my $label2 = $ptypesmap{$_->[0]};
    my $bond_order = $_->[1];

    # Unless the next door neighbor is a carbon, and is bonded
    # to this carbon with a bond order of 1.5, skip this iteration
    unless ($label2 =~ /^C/ and $bond_order == 1.5) {
      next;
    }

    my @labels = sort ($label1, $label2);
    $outhash{"X $labels[0] $labels[1] X 6 25 0 -25 0 0 0 0" .
          "  # Generic torsional parameters"} = 1;
    }
  }
  }

  foreach (keys %outhash ) {
  $ret .= "$_\n";
  }

  $ret .= "# end\n";
  $ret .= "#\n";

  return $ret;
}


# Output blank torsions for all of the other torsions not found in the
# generic torsions.  We could have saved the data from the previous
# section, but instead of carrying around more data, we'll simply
# recalculate it here.
sub print_other_blank_torsions(\%)  {
  my $molref = shift;
  my %mol = %{$molref};
  my $ret;  # For return string
  my %ring_torsions;
  my %outhash;

  $ret .= "#\n";
  $ret .= "# torsion\n";

  # We'll need a reversed version of the types hash to work with.
  my %ptypesmap;
  foreach ( keys( %{$mol{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$mol{types}{$_}};
  foreach (@atomlist) {
    $ptypesmap{$_} = $type;
  }
  }

  # We're only generating generic torsions for pairs of aromatic
  # carbons.  A carbon is aromatic if it has three neighbors, and at
  # least two bonds that are of bond order 1.5.
  foreach (keys( %{$mol{types}} ) ) {
  # For each of these, we'll only look at the first atom in the
  # given list, since others should be suitably similar.  This will
  # generate the blacklist for all types with identical carbons.
  # Note that some types are identical (because of identical
  # qcodes), and will not explicitly be covered here.
  my $this_atom = $mol{types}{$_}[0];
  my $this_type = $_;
  my $resonant_bond_count;

  # If it's not a carbon, just skip to the next one.
  unless ($this_type =~ /^C/ ) {
    next;
  }

  my @connectivity = @{$mol{connectivity}{$this_atom} };

  $resonant_bond_count = 0;
  if (scalar(@connectivity ) == 3 ) {
    # This atom may be aromatic, let's look through the connectivity
    # to find out.
    foreach (@connectivity) {
    my $bond_order = $_->[1];
    my $other_type = $ptypesmap{$_->[0]};

    if ($bond_order == 1.5 and $other_type =~ /^C/) {
      $resonant_bond_count++;
    }
    }
  }

  if ($resonant_bond_count >= 2) {
    # If we've got an aromatic carbon, and neighbors, we simply add
    # the line to our ring_torsions.  The key will be the entire output
    # line, and the value will be simply 1.  Once again, we sort the
    # 2 labels so we don't get duplicate lines.
    my $label1 = $this_type;

    foreach (@connectivity) {
    my $label2 = $ptypesmap{$_->[0]};
    my $bond_order = $_->[1];

    # Unless the next door neighbor is a carbon, and is bonded
    # to this carbon with a bond order of 1.5, skip this iteration
    unless ($label2 =~ /^C/ and $bond_order == 1.5) {
      next;
    }

# Display the blacklisted atoms, for troubleshooting.
#if ($_->[0] <$this_atom ) {
#print "" . ($_->[0] + 1) . " " . ($this_atom + 1) . "\n";
#} else {
#print "" . ($this_atom + 1) . " " . ($_->[0] + 1) . "\n";
#}

    my @labels = sort ($label1, $label2);
    $ring_torsions{"$labels[0] $labels[1]"} = 1;
    }
  }
  }

  # Ok, we now have what is more or less a 'blacklist' of torsions
  # already done.  Now generate a hash full of all possible torsions.
  # We'll then go through that list, and remove all members that match
```

```
  # something from the original hash.  This is an interesting problem,
  # since we're actually doing 4 nested loops in order to get all of them.
  foreach (keys( %{$mol{types}})) {
  my $this_atom = $mol{types}{$_}[0];
  my $this_type = $_;
  my $atom2 = $this_atom;
  my ($atom1, $atom4);

  my @conn1 = @{$mol{connectivity}{$atom2} };
  # If this atoms isn't connected to at least 2 things, it can't
  # possibly be a central atom for a dihedral.
  if (scalar(@conn1) < 2) {
    next;
  }

  my ($i, $j, $k);
  for ($i = 0; $i <= $#conn1; $i++) {
    # Now we look for another atom that might be a central atom.
    my $atom3 = $conn1[$i][0];
    my @conn2 = @{$mol{connectivity}{$atom3}};

    # Once again, we see if this could possibly be a central atom.
    if (scalar(@conn2) < 2) {
    next;
    }

    # If we got here, then we're ready to roll, loop over all of the
    # other atoms from @conn1, and loop over all of the non-$atom2
    # atoms in @conn2.
    for ($j = 0; $j <= $#conn1; $j++) {
    if ($j == $i) {
      next;
    }
    for ($k = 0; $k <= $#conn2; $k++) {
      if ($conn2[$k][0] == $atom2) {
      next;
      }
      # It is in this body that we generate the entire torsion.
      # There are a lot of players, so move carefully.
      $label1 = $ptypesmap{$conn1[$j][0]};
      $label2 = $this_type;
      $label3 = $ptypesmap{$conn1[$i][0]};
      $label4 = $ptypesmap{$conn2[$k][0]};

      # We need to see if we have to reverse the labels.  We will
      # use the standard string comparison to do so.
      if ($label2 gt $label3) {
      my $tmp;
      $tmp = $label1;
      $label1 = $label4;
      $label4 = $tmp;

      $tmp = $label2;
      $label2 = $label3;
      $label3 = $tmp;
      }

      # Now we're ready to put the information in the %outhash;
      $outhash{"$label1 $label2 $label3 $label4"} = 1;
    }
    }
  }
  }

# To see the ring torsions, uncomment the following section;
#foreach (keys %ring_torsions ) {
#  print "Key $_, Value $ring_torsions{$_}\n";
#}

  # We've got our comprehensive list, and our blacklist.  Clear out
  # the comprehensive list;
  my %work_hash = %outhash;
  foreach (keys %work_hash ) {
  my (undef, $label1, $label2, undef) = split(" ", $_);
  if (exists($ring_torsions{"$label1 $label2"})) {
    delete $outhash{$_};
  }
  }

# Uncomment the following section to get a comprehensive list of all
# torsions that are checked.  Note that symmetry related torsions are
# not included in the list.
#my %numhash;
#foreach (keys %outhash ) {
#  my ($lab1, $lab2, $lab3, $lab4) = split;
#  my ($at1, $at2, $at3, $at4) = ( ($abs_parent{types}{$lab1}->[0] + 1),
#                  ($abs_parent{types}{$lab2}->[0] + 1),
#                  ($abs_parent{types}{$lab3}->[0] + 1),
#                  ($abs_parent{types}{$lab4}->[0] + 1) );
#  $numhash{"$at1 $at2 $at3 $at4"} = 0;
#}
#foreach (sort keys %numhash) {
#  print "$_\n";
#}


  # That's it, return the rest.
  foreach (keys %outhash ) {
  $ret .= "$_ 6 0.0 0.0 0.0 0.0 0.0 0.0 0.0\n";
  }

  $ret .= "# end\n";
  $ret .= "#\n";

  return $ret;
}

# This function call will only be valid for the hash that contains the
# absorbed children.  It will attach a 'frozen' key to that hash which
# has all of the bonds that must be frozen, along with the proper
# translation to it's own numbering system.
sub attach_frozen_keys(\%) {

  my $molref = shift;

  my %this_mol = %{$molref};

  foreach (keys %this_mol ) {
  my $dir = $_;
  my $fullpath = "$db_path/$dir";

  open(FROZEN, "<$fullpath/Frozen_bonds") or
    die "Unable to open $fullpath/Frozen_bonds for reading";

  my @freezelist = <FROZEN>;
  chomp @freezelist;

  close(FROZEN);

  # Do the translation to the numbering system used by the absorbed
  # children, since the 'raw' list is provided in terms of the
  # unabsorbed children.  Since the list only marks 'bonds' as
  # frozen, we'll also end up doing a search to provide all of the
  # Possible frozen dihedrals.  This will be done after the mapping
  # is done.
  foreach (@freezelist) {
    my ($atom2, $atom3) = split(" ", $_);

    # Since the current atom numbers are from the fragments, we need
    # to look at the tosister field of the full fragments.
    $atom2 = $children{$dir}{tosister}{$atom2};
    $atom3 = $children{$dir}{tosister}{$atom3};

    # And put the transcribed values back on the list
    $_ = "$atom2 $atom3";
  }

  # Use the provided values to generate all possible dihedrals.
  my @worklist;
  my ($i, $j);
  foreach (@freezelist) {
    my ($atom2, $atom3) = split(" ", $_);

    # Loop over all atoms on either side of the bond, and enter
    # them.
    my @conn1 = @{$this_mol{$dir}{connectivity}{$atom2} };
    my @conn2 = @{$this_mol{$dir}{connectivity}{$atom3} };

    for ($i = 0; $i <= $#conn1; $i++ ) {
    my $atom1 = $conn1[$i][0];

    if ($atom1 == $atom3) {
      next;
    }
    for ($j = 0; $j <= $#conn2; $j++ ) {
      my $atom4 = $conn2[$j][0];
      if ($atom4 == $atom2) {
      next;
      }
      # If we got here, we must have a real torsion to report, do
      # it.

      # Before we actually freeze these bonds, we also need to
      # know what value they were frozen at.  This can be
      # retrieved from the Initial_optimization.log file.
      # Remember, the current $atom1, $atom2, etc. are all
      # for the absorbed member, we need the dihedral in the
      # unabsorbed fragment.
      my @dihedral = (
          $this_mol{$dir}{tosister}{$atom1},
          $this_mol{$dir}{tosister}{$atom2},
          $this_mol{$dir}{tosister}{$atom3},
          $this_mol{$dir}{tosister}{$atom4},
          );
      my ($retval, undef) = split ( " ",
      get_last_dihedral_and_energy(
        "$db_path/$dir/Initial_optimization.log",
        @dihedral
                  )
                  );
      push(@worklist, "$atom1 $atom2 $atom3 $atom4 $retval");
    }
    }
  }

  # Worklist now has all of the frozen torsions, so we'll just copy
  # it to freezelist, so we're ready to put them into the parent
  # molecule.

  @freezelist = @worklist;

  # Add the field.
  ${$molref}{$dir}{frozen} = [ @freezelist ];
  }
}

# Once again, we have bad design being used to speed prototyping.
# This function takes no arguments, since it reads all of the things
# it needs from existing global arguments.  It will create the
# appropriate directory structure to finish the force fields with
# Matt's torsion fitting programs.

# Based on information from Matt (Morning, 4/15/02), we need in the
# master directory to provide parameters for each and every atom types
# we ever encounter.

sub make_dir_structure {

  # Temporarily simply delete the whole spiel, until we are to
  # production.
  if ($cleandir) {
  system("rm -rf ./myff");
  }

  # Create our directory structure.  Don't create it if it exists,
  # just warn the user and exit.  This has been changed.  This program
  # will refresh the files it knows about in the directory based on
  # the answer to a question asked in the beginning.
  unless ($cleandir) {
```

```
#  if ( -e "./myff") {
#    print "File (or directory) ./myff already exists.  Will not " .
#        "continue.  If you wish to create the file structure, " .
#      "please delete or rename the file (or directory)";
#    exit;
#  }

  } else {
  mkdir "./myff" or die "Unable to create directory ./myff";
  }

  chdir "./myff" or die "Unable to change directory to ./myff";

  # Make all necessary sub directories.
  my @dirlist = (keys %abs_children);

  foreach ( @dirlist ) {
  unless (-e $_ ) {
    mkdir $_ or die "Unable to create directory $_";
  }
  }

  # Create a fragment_structures directory, and populate it with .com
  # files that can be read by gaussview

  unless (-e './fragment_structures') {
  mkdir './fragment_structures';
  }

  open(COM, ">./fragment_structures/abs_parent.com") or
  die "Unable to open /fragment_structures/abs_parent.com for writing";
  select COM;
  make_com_file( %abs_parent );
  close COM;
  select STDOUT;

  foreach ( keys %abs_children ) {
  my $dir = $_;
  my %fragment = %{$abs_children{$dir}};

  open(COM, ">./fragment_structures/$dir.com") or
    die "Unable to open ./fragment_structures/$dir.com for writing";
  select COM;
  make_com_file( %fragment );
  close COM;
  select STDOUT;
  }


  # For each of the directories, populate it with a proper .str file.
  # Since these are all 'pre-simulation' fragments, we'll simply put
  # the absorbed fragments in.  This program has the data for
  # translation at a later point, if necessary.

  # According to Matt's latest e-mail, (Morning, 4/15/02) we _don't
  # need to provide mff files for the children, only for the parent.
  foreach (@dirlist) {
  my $this_dir = $_;
  my $this_str;
  my $this_mff;

    $this_str = $abs_children{$this_dir}{str};
    $this_mff = $abs_children{$this_dir}{mff};

  open(STR, ">$this_dir/STR.str") or die "Unable to open " .
    "$this_dir/STR.str for writing";
  print STR $this_str . "\n";
  close STR;
  }

  # Ok, all of the .str and .mff files have been created.  Now add the
  # 'master' directory, and also put in all of the parameter files.
  unless (-e 'master' ) {
  mkdir "master" or die "Unable to create master directory";
  }

  # And add an .str file for the parent.
  open STR, ">./master/master.str" or
  die "Unable to open ./master/master.str for writing";
  print STR $abs_parent{str};
  close STR;

  # Params is a list of file prefixes, and the associated keyword in
  # the associated .mff file.  Note that there may be multiple torsion
  # sections, and this implementation will take only the first torsion
  # section.  The first member of the list at each key is the name to
  # search the file on, the second member is the number of relevant
  # label terms.
  my %params = (
        gen_tors => ['torsion', 4 ],
                        stretch  => ['stretch', 2 ],
                        bend     => ['bend', 3],
        mass     => ['mass', 1 ],
        vdw      => ['vdw', 2 ],
        inv      => ['inversion', 4 ],
        );

  # This section is a bit trickier.  We need to search all of the .mff
  # files, and store the proper section into a hash.  They keys will
  # be the labels seciton, and the value will be the rest of the line.
  foreach (keys %params) {
  my $filetype = $_;
  my $filename = "master/$_.params";
  my $searchkey = "# $params{$_}->[0]";
  my $labelcount = $params{$_}->[1];
  my $contents;

  my $searchstring = "$searchkey.*?^(.*?)# end";

  # We'll now search the parent, and all of the children for
  # parameters of this type
  unless($abs_parent{mff} =~ /$searchstring/sm) {
    die "Failed to match $searchstring in mff of abs_parent";
  };
  $contents = $1;

  foreach (keys %abs_children ) {
    my %this_frag = %{$abs_children{$_}};
    unless($this_frag{mff} =~ /$searchstring/sm) {
    die "Failed to match $searchstring in mff of abs_children in " .
        "directory $_";
    };
    $contents .= $1;
  }

  # Now, take the contents and hash them, so we don't have any
  # repeats.
  my @contentlist = split("\n", $contents);

  my %compressed_contents;
  foreach (@contentlist) {
    my $line = $_;
    my @splitvals = split(" ", $line, ($labelcount + 1));

    my $value = pop @splitvals;
    my $key = join(" ", @splitvals);

    unless (exists($compressed_contents{$key})) {
    $compressed_contents{$key} = $value;
    } else {
    # This indicates the condition where we have more than one
    # entry for the given atom types.  This should be common in
    # all of the cases, but if we end up mapping more unique
    # values later, we may want to use this section for that test.

    # print "Warning, duplicate key detected in $filetype\n";
    }
  }


  # Now just print it to its proper file.
  open(PRM, ">$filename") or die "Unable to open $filename for writing";
  foreach (keys(%compressed_contents)) {
    print PRM "$_ $compressed_contents{$_}\n";
  }
  close PRM;
  }

  # We will also need to collect all of the torsions from all of the
  # children, as well, as these will go into a file called
  # all_torsions.params, and will include the torsions from the
  # parent, and all of the children.

  my @contentlist = split "\n", $abs_parent{mff};
  # Trim the front of list till we find what we're looking for;
  my $pattern = '[A-Z][\w\d]+ [A-Z][\w\d]+ [A-Z][\w\d]+ [A-Z][\w\d]+ ' .
            '[\d\s.]+';
  until ($contentlist[0] =~ /^$pattern$/) {
  shift @contentlist;
  }

  # Trim the back of the list similarly
  until ($contentlist[-1] =~ /^$pattern$/) {
  pop @contentlist;
  }

  my @alltorsions = @contentlist;

  # Now repeat the previous for all of the absorbed children
  foreach (keys %abs_children ) {
  @contentlist = split "\n", $abs_children{$_}{mff};
  until ($contentlist[0] =~ /^$pattern$/) {
    shift @contentlist;
  }

  # Trim the back of the list similarly
  until ($contentlist[-1] =~ /^$pattern$/) {
    pop @contentlist;
  }

  push @alltorsions, @contentlist;
  }

  # Hash it for uniqueness, then write the file out
  my %torsions;
  $pattern = '([A-Z][\w\d]+ [A-Z][\w\d]+ [A-Z][\w\d]+ [A-Z][\w\d]+) ' .
          '([\d\s.]+)';

  foreach (@alltorsions) {
  $_ =~ /^$pattern$/ or die "Unable to reproduce match while building ".
                          "all torsions file with line:\n$_\n";
  $torsions{$1} = $2;
  }

  open TORS, ">master/all_torsions.params" or
  die "Unable to open master/all_torsions.params for writing";
  foreach (sort keys %torsions ) {
  print TORS "$_ $torsions{$_}\n";
  }
  close TORS;

  # Output our atom_map_list and bond_map_list so we know which
  # fragments are 'responsible' for providing which parameters.
  my $i;
  open MAP, ">./master/atom_map_list" or
  die "Unable to open ./master/atom_map_list for writing";
  print MAP <<HEADER;
The following file provides the mapping that is used for bond to
bond correlations between the parent and the child fragments.  The atom
numbers provided correspond to the _gaussview_ numbers.  If you are
looking for the appropriate numbers in a .create or .verify files, you
must subtract one from the numbers provided.  All numbers are for the
appropriate absorbed fragments, which can be found in
../fragment_structures/.

HEADER
```

```
  for ( $i = 0; $i <= $#atom_map_list; $i++ ) {
  my $upatom = $i;
  my ($fragment, $ufatom) =
    ( $atom_map_list[$upatom]{directory},
    $atom_map_list[$upatom]{fragatom} );
  unless (exists $parent_molecule{tosister}{$upatom}) {
    next;
  }

  my $apatom = $parent_molecule{tosister}{$upatom};
  my $afatom = $children{$fragment}{tosister}{$ufatom};
  print MAP "Parent atom " . ($apatom+1) .
           " -> $fragment " . ($afatom + 1) .
       " -> qdb atom " . ($ufatom + 1) . "\n";
  }
  close MAP;

  # Now output the bond map list.  This section is only slightly more
  # tedious, but not difficult to follow.
  open MAP, ">./master/bond_map_list" or
  die "Unable to open ./master/bond_map_list for writing";
  print MAP <<HEADER;
The following file provides the mapping that is used for bond to
bond correlations between the parent and the child fragments.  The atom
numbers provided correspond to the _gaussview_ numbers.  If you are
looking for the appropriate numbers in a .create or .verify files, you
must subtract one from the numbers provided.  All numbers are for the
appropriate absorbed fragments, which can be found in
../fragment_structures/.

HEADER

  # The (somewhat wordy) anoymous sort routine sorts the output of the
  # bonds.
  foreach (sort {
  ($a =~ /(\d+)-(\d+)/);
  $a1 = $1; $a2 = $2;
  $b =~/(\d+)-(\d+)/;
  $b1 = $1; $b2 = $2;
  unless ($a1 == $b1) {
    $a1 <=> $b1;
  } else {
    $a2 <=> $b2;
  }
  } keys %bond_map_list ) {
  my ( $upatom1, $upatom2 ) = split("-", $_);
  my ( $ufatom1, $ufatom2 ) = split("-", $bond_map_list{$_}{fragbond});
  my $fragment = $bond_map_list{$_}{directory};

  unless (exists $parent_molecule{tosister}{$upatom1} and
      exists $parent_molecule{tosister}{$upatom2} ) {
    next;
  }

  my ( $apatom1, $apatom2 ) = ( $parent_molecule{tosister}{$upatom1},
                 $parent_molecule{tosister}{$upatom2});
  my ( $afatom1, $afatom2 ) = ( $children{$fragment}{tosister}{$ufatom1},
                 $children{$fragment}{tosister}{$ufatom2});

  print MAP "Parent bond " . ($apatom1 + 1) . "-" . ($apatom2 + 1) .
        " -> $fragment " . ($afatom1 + 1) . "-" . ($afatom2 + 1) .
      " -> qdb bond " . ($ufatom1 + 1) . "-" . ($ufatom2 + 1) . "\n";

  }

  close MAP;


  # We now create the master.mff file.  It turns out this task is almost
  # trivial, since we've written the information already into the
  # Individual files

  $this_mff = '';
  my @fieldlist = ('mass.params mass', 'vdw.params vdw',
       'stretch.params stretch', 'bend.params bend',
       'inv.params inversion', 'gen_tors.params torsion',
       'all_torsions.params torsion');

  foreach ( @fieldlist ) {
      my ($filename, $keyword) = split (' ', $_);
      open TMP, "<master/$filename" or
    die "Unable to open master/$filename for reading";
      my @allofit = <TMP>;
      close TMP;
      $this_mff .= "#\n# $keyword\n";
      $this_mff .= join('', @allofit);
      $this_mff .= "# end\n"#\n";
  }

  # If there is a 'completed_torsions' file, we need to read it, and
  # append those values to $this_mff as well.
  if (-e "master/completed_torsions") {

  # First, get our mapping of fragment to parent torsions.
  my %ftop;
  open TMP, "<master/frag_to_parent_torsions" or
    die "Unable to open master/frag_to_parent_torsions";
  while (<TMP>) {
    chomp $_;
    my @values = split(" ", $_);
    my $key = join(" ", @values[1..4]);
    my $value = join(" ", @values[5..8]);

    $ftop{$key} = $value;
  }
  close TMP;

  # The first task is to get the information into a hash.
  my %completed_torsions = ();
  open TMP, "<master/completed_torsions" or
    die "Unable to open master/completed_torsions for reading";
  while (<TMP>) {
    chomp;
    my ($dir, $num, $tors, $params) = split(",", $_);
    $completed_torsions{$tors} = $params;
  }
  close TMP;

  # Condition the existing $this_mff for a new torsion section
  $this_mff .= <<TORS;
#
# These torsions have been declared finished from a previous run of
# this program.
# torsion
TORS

  # Now we find the mappings, so we can add parameters for both the
  # children and the parent.
  foreach (keys %completed_torsions ) {
    my $tors = $_;
    my $params = $completed_torsions{$tors};

    unless (exists ( $ftop{$tors} )) {
    die "Unable to find parent torsion for child torsion " .
        "$tors.  This can only happen if we screwed " .
      "up some accounting earlier.  Cannot continue";
    }
    $completed_torsions{$ftop{$tors}} = $params;
  }

  # Beautiful, we have all the lines we need, just stick them onto
  # the end of $this_mff.
  foreach (keys %completed_torsions ) {
    my $tors = $_;
    my $params = $completed_torsions{$tors};

    $this_mff .= "$tors $params\n";
  }



  $this_mff .= <<TORS;
# end
TORS

  }

  open(MFF, ">master/master.mff") or die "Unable to open " .
  "master/master.mff for writing";
  print MFF <<INTRO;
# This master file has all parameters for every torsion that exists,
# both in the parent file, and in all of the children.  It contains
# exactly one of each parameter (There are no repeats).
INTRO

  print MFF $this_mff;
  close MFF;


  # We'll need a struct_names file as well.  This file starts with the
  # number of entries that there will be, and continues by providing
  # relative paths on each line to the individual fragments.
  my @substructures;
  foreach (keys %abs_children ) {
  push(@substructures, "../$_/STR.str");
  }

  open(STRUCT, ">master/struct_names") or die "Unable to open " .
  "master/struct_names for writing";
  print STRUCT scalar(@substructures) . "\n";
  print STRUCT join("\n", @substructures) . "\n";
  close STRUCT;

  # Now we need to put the actual torsion information into each of the
  # appropriate directories.  While we're doing this, we'll also need
  # to create the 'map' file which is a simple text file that says
  # which child torsions are mapped to which parent torsions.  I used
  # to gene
  my %f_to_par_map;
  foreach (keys %abs_children ) {
  my $dir = $_;
  my $tindex = 0;

  # If there's no torsions for this directory, move on silently
  unless (exists($abs_children{$_}{torsions})) {
    next;
  }

  foreach (@{$abs_children{$dir}{torsions}}) {
    my %torsion = %{$_};

    # As a refresher, the relevant structure of this is:
      # fatomnums       => 11 12 13 14
      # fatomtypes      => O_3 C24 C215 C222
      # patomnums       => 35 43 44 45
      # patomtypes      => O_3 C24 C215 C222
      # angle_vs_energy => <some complete file>

    # Create our angle_vs_energy file;
    open TMP, ">$dir/angle_vs_energy_$tindex" or
    die "Unable to open $dir/angle_vs_energy_$tindex for writing";
    print TMP $torsion{angle_vs_energy};
    close TMP;

    # Generate a list of all torsions around the 'interesting one'
    my @alltorsions;
    my (undef, $atom2, $atom3, undef) = split(" ", $torsion{fatomnums});
    my @conn1 = @{$abs_children{$dir}{connectivity}{$atom2}};
    my @conn2 = @{$abs_children{$dir}{connectivity}{$atom3}};

    foreach (@conn1) {
    my $atom1 = $_->[0];
    if ($atom1 == $atom3) {
      next;
    }
    foreach (@conn2) {
      my $atom4 = $_->[0];
```

```
  if ($atom4 == $atom2) {
  next;
  }
  push @alltorsions, "$atom1 $atom2 $atom3 $atom4";
}
}

# Start with our accounting for mapping
$f_to_par_map{"$dir $torsion{fatomtypes} $torsion{patomtypes}"} = 1;

# As it turns out, we occasionally have to fit several torsions
# at once.  The previous line only records the 'driven' torsion.
# We also need to record all other torsions before proceeding.

# Before we proceed, we need to reverse the qcode hashes for
# both the parents, and children.  We'll make the hashes refer
# directly to the atom numbers on the absorbed species, and
# the values will be the already split lists.
my %parent_qcodes;
my %fragment_qcodes;
foreach (keys %{$parent_molecule{qcodes}} ) {
my @qcode = split;
my @atomlist = @{$parent_molecule{qcodes}{$_}};
foreach (@atomlist) {
  my $unabsatom = $_;
  my $absatom = $parent_molecule{tosister}{$unabsatom};
  unless (defined $absatom) {
  next;
  }
  # So now we can make our entry
  $parent_qcodes{$absatom} = [@qcode];
}
}

# And do the same for this particular fragment
foreach (keys %{$children{$dir}{qcodes}} ) {
my @qcode = split;
my @atomlist = @{$children{$dir}{qcodes}{$_}};
foreach (@atomlist) {
  my $unabsatom = $_;
  my $absatom = $children{$dir}{tosister}{$unabsatom};
  unless (defined $absatom) {
  next;
  }
  # So now we can make our entry
  $fragment_qcodes{$absatom} = [@qcode];
}
}

my %dont_use;
foreach (@alltorsions) {
if ($torsion{fatomnums} eq $_) {
  $dont_use{$_} = 1;
  next;
}

if (exists($dont_use{$_})) {
  next;
}

# Now, similar to how we generated the torsion hash in the
# first place, find the 'best match' to the current torsion.
# For each terminus on the fragment torsion, we are looking
# for a match in the parten.
my ($fatom1, $fatom2, $fatom3, $fatom4) = split;
my ($patom1, $patom2, $patom3, $patom4) =
  split(" ", $torsion{patomnums});

# Find the match for one end
my @conn1 = @{$abs_parent{connectivity}{$patom2}};
my @qcode1 = @{$fragment_qcodes{$fatom1}};

# Do our double loop search;
my $best_match = undef;
foreach (@conn1) {
  my $test_patom1 = $_->[0];
  if ($test_patom1 == $patom3) {
  next;
  }

  # Get our qcode;
  my @qcode2 = @{$parent_qcodes{$test_patom1}};


  # Now check our match;
  my $match = CFUNCS::get_qcode_deviance( \@qcode1, \@qcode2);
  unless (defined $best_match) {
  $best_match = $match;
  $patom1 = $test_patom1;
  }

  if ($match > $best_match) {
  $best_match = $match;
  $patom1 = $test_patom1;
  }
}

# And now get the match on the other end.
@conn1 = @{$abs_parent{connectivity}{$patom3}};
@qcode1 = @{$fragment_qcodes{$fatom4}};

# Do our double loop search;
$best_match = undef;
foreach (@conn1) {
  my $test_patom4 = $_->[0];
  if ($test_patom4 == $patom2) {
  next;
  }

  # Get our qcode;
  my @qcode2 = @{$parent_qcodes{$test_patom4}};


  # Now check our match;
  my $match = CFUNCS::get_qcode_deviance( \@qcode1, \@qcode2);
  unless (defined $best_match) {
  $best_match = $match;
  $patom4 = $test_patom4;
  }

  if ($match > $best_match) {
  $best_match = $match;
  $patom4 = $test_patom4;
  }
}

# Now that we have the numbers, getting the labels is trivial.
# Instead of reversing the hash, as we normally do, we'll just
# search through it.
my ($plabel1, $plabel2, $plabel3, $plabel4) =
  split(" ", $torsion{patomtypes});
my ($flabel1, $flabel2, $flabel3, $flabel4) =
  split(" ", $torsion{fatomtypes});

# And undef the ones we don't know yet.
$plabel1 = $plabel4 = $flabel1 = $flabel4 = undef;

foreach (keys %{$abs_parent{types}} ) {
  if (defined $plabel1 and defined $plabel4) {
  last;
  }
  my $label = $_;
  my @atomlist = @{$abs_parent{types}{$label}};
  foreach (@atomlist) {
  my $testatom = $_;
  if ($testatom == $patom1) {
    $plabel1 = $label;
    next;
  }
  if ($testatom == $patom4) {
    $plabel4 = $label;
    next;
  }
  }
}

# Do the same for the fragment atoms
foreach (keys %{$abs_children{$dir}{types}} ) {
  if (defined $flabel1 and defined $flabel4) {
  last;
  }
  my $label = $_;
  my @atomlist = @{$abs_children{$dir}{types}{$label}};
  foreach (@atomlist) {
  my $testatom = $_;
  if ($testatom == $fatom1) {
    $flabel1 = $label;
    next;
  }
  if ($testatom == $fatom4) {
    $flabel4 = $label;
    next;
  }
  }
}

# And finally, add the information.
my $parentlabels = "$plabel1 $plabel2 $plabel3 $plabel4";
  my $fragmentlabels = "$flabel1 $flabel2 $flabel3 $flabel4";
$f_to_par_map{"$dir $fragmentlabels $parentlabels"} = 1;

}


# Make a hash version of %alltorsions for quick searching;
my %alltorsions;
foreach (@alltorsions) { $alltorsions{$_} = 1; }

# Ok, this code is getting pretty schlocky (is that even a
# word?).  We need to establish %alltorsions before removing the
# retraints, but _after_ we've removed the restraints, we need
# to get rid of duplicate labels.  Oh well.

# Remember that when we print out frozen bonds, we _don't_ want
# to print out bonds that we are driving, as it makes no sense
# to freeze those.  Initialize a hash that we'll use for those
# sections.
my %restraints = ();
foreach ( @{$abs_children{$dir}{frozen}} ) {
my @tmp = split(" ", $_);
my $chkline = join(" ", @tmp[0..3] );
unless ( exists($alltorsions{$chkline}) ) { $restraints{$_} = 1; }
}

my $numrestraints = scalar( keys %restraints );

# Find the maximum symmetry about the given torsion, so we can
# request those constraints from Matt's programs
my (undef, $tatom1, $tatom2, undef) = split(" ", $torsion{patomnums});
my $symmetry = find_max_symmetry( %abs_parent, $tatom1, $tatom2 );

# Finally, as it turned out, @alltorsions and %alltorsions
# contained 'extra' bonds, where there were some of the same
# atom types.  This caused problems in the fitting, so we need
# to remove those that have 'registered' atom types.

# First, find the types that are 'already' taken.
my %used_labels;
my $labelstring = $torsion{fatomtypes};
$used_labels{$labelstring} = 1;

# We'll need the reversed types map
my %ftypesmap;
foreach ( keys( %{$abs_children{$dir}{types}} ) ) {
my $type = $_;
my (@atomlist) = @{$abs_children{$dir}{types}{$_}};
foreach (@atomlist) {
  $ftypesmap{$_} = $type;
```

```
    }
    }

    foreach (keys %alltorsions ) {
        # We don't want to delete our primary torsion.
        if ($_ eq $torsion{fatomnums}) {
      next;
        }

        # All we do here is get the labelstring that corresponds to
        # our torsion string, if it exists, delete this torsion.

    my ($atom1, $atom2, $atom3, $atom4) = split(" ", $_);
        my $labelstring = "$ftypesmap{$atom1} $ftypesmap{$atom2} " .
      "$ftypesmap{$atom3} $ftypesmap{$atom4}";

    if (exists $used_labels{$labelstring}) {
      delete $alltorsions{$_};
    } else {
      $used_labels{$labelstring} = 1;
    }
    }

    # And put these back into the list
    @alltorsions = keys %alltorsions;

    my $numtorsions = scalar(keys %alltorsions);


# Uncomment the following to see the assigned symmetries
#print "For bond: " . ($tatom1 + 1) . "-" . ($tatom2 + 1) . " in the " .
#      "absorbed parent (dir $dir), the symmetry was $symmetry\n";

    # Create the fit file.
    open TMP, ">$dir/torsion_$tindex.create" or
    die "Unable to open $dir/fit_$tindex.min_par for writing";
    print TMP <<INP;
# amber_flag ff_file
0 ./master.mff
# config_file
../$dir/STR.config
# torsion_file
../$dir/angle_vs_energy_$tindex
# min_switch
2
# n_driven
1
# (driven_atom_1[i] driven_atom_2[i] driven_atom_3[i] driven_atom_4[i]
#  driven_dihed_1[i] (degrees) driven_dihed_2[i] (degrees)
#  n_min_interv[i]), i = 0, n_driven - 1
$torsion{fatomnums} 0.0 360.0 72
# n_fitted
$numtorsions
# (fitted_atom_1[i] fitted_atom_2[i] fitted_atom_3[i] fitted_atom_4[i]
#  min_fit_order[i] min_sym[i]), i = 0, n_fitted - $numtorsions
INP
foreach ( @alltorsions ) {
  print TMP "$_ 6 $symmetry\n";
}

print TMP <<INP;
# n_restraints
$numrestraints
# (restraint_atom_1[i] restraint_atom_2[i] restraint_atom_3[i]
restraint_atom_4[i]
#  restraint_dihed[i] (degrees)), i = 0, n_restraints - $numrestraints
INP

foreach ( sort keys %restraints ) {
  print TMP "$_\n";
}

print TMP <<INP;
# min_tol k_restraint (kcal/mole/degrees^2)
1.0e-10 10.0
# k_cell
1
# r_on (angstroms) r_off (angstroms) skin (angstroms)
10.0 12.0 40.0
# long_range_switch coul_switch ewald_switch vdw_switch
0 0 0 2
# vdw_rad_switch one_four_switch bend_switch
0 0 0
# ewald_accuracy delta_grid (angstroms) B_spline_order
1.0e-4 1.0 6
# scale_1_4_flag scale_1_4_vdw scale_1_4_coul
0 1.0 1.0
# exo_switch exo_scale
0 1.0
# orient_switch orient_file
0 no_file
# elong_switch elong_file
0 no_file
# elect_switch elect_field (V/micron) elect_vect.x elect_vect.y elect_vect.z
0 3000.0 0.0 1.0 0.0
# pore_switch pore_sigma (angstrom) pore_epsilon (kcal/mol) pore_r0 (angstrom)
pore_z0 (angstrom)
0 3.0 0.1 5.0 10.0
# posit_switch
1
# bond_switch angle_switch dihed_switch
0 0 0
# graph_switch graph_stride
1 10
INP
    close TMP;

    # The .create fine is finished, get the verify file written

    # Create the check file.
    open TMP, ">$dir/torsion_$tindex.verify" or
    die "Unable to open $dir/fit_$tindex.min_par for writing";
    print TMP <<INP;
# amber_flag ff_file
0 ./master.mff
# config_file
../$dir/STR.config
# torsion_file
../$dir/angle_vs_energy_$tindex
# min_switch
1
# n_driven
1
# (driven_atom_1[i] driven_atom_2[i] driven_atom_3[i] driven_atom_4[i]
#  driven_dihed_1[i] (degrees) driven_dihed_2[i] (degrees)
#  n_min_interv[i]), i = 0, n_driven - 1
$torsion{fatomnums} 0.0 360.0 72
# n_fitted :
0
# (fitted_atom_1[i] fitted_atom_2[i] fitted_atom_3[i] fitted_atom_4[i]
#  min_fit_order[i] min_sym[i]), i = 0, n_fitted - 0
# n_restraints
$numrestraints
# (restraint_atom_1[i] restraint_atom_2[i] restraint_atom_3[i]
restraint_atom_4[i]
#  restraint_dihed[i] (degrees)), i = 0, n_restraints - $numrestraints
INP

foreach ( sort keys %restraints ) {
  print TMP "$_\n";
}

print TMP <<INP;
# min_tol k_restraint (kcal/mole/degrees^2)
1.0e-10 10.0
# k_cell
1
# r_on (angstroms) r_off (angstroms) skin (angstroms)
10.0 12.0 40.0
# long_range_switch coul_switch ewald_switch vdw_switch
0 0 0 2
# vdw_rad_switch one_four_switch bend_switch
0 0 0
# ewald_accuracy delta_grid (angstroms) B_spline_order
1.0e-4 1.0 6
# scale_1_4_flag scale_1_4_vdw scale_1_4_coul
0 1.0 1.0
# exo_switch exo_scale
0 1.0
# orient_switch orient_file
0 no_file
# elong_switch elong_file
0 no_file
# elect_switch elect_field (V/micron) elect_vect.x elect_vect.y elect_vect.z
0 3000.0 0.0 1.0 0.0
# pore_switch pore_sigma (angstrom) pore_epsilon (kcal/mol) pore_r0 (angstrom)
pore_z0 (angstrom)
0 3.0 0.1 5.0 10.0
# posit_switch
1
# bond_switch angle_switch dihed_switch
0 0 0
# graph_switch graph_stride
1 10
INP
    close TMP;
    # End matt's .create and .verify files.

    # Now make the config files with build_single (Matt's programs
    # must be in the users path), but only if $tindex == 0, since it
    # only needs to be done once per directory.
    if ($tindex == 0) {
    # Note that we say and die, since system commands normally
    # return 0 upon success.
    system("build_single $dir/STR.str $dir/STR.config") and
      die "Trouble running commmand:\n" .
        "build_single $dir/STR.str $dir/STR.config";
    }

    # We will only create the directory structure here.  After it's
    # finished, we'll go into interactive mode to do the rest of the
    # fitting.

    $tindex++;
  }
  }

  # Print out the torsions maps into the master directory.
  open TMP, ">master/frag_to_parent_torsions" or
  die "Unable to open master/frag_to_parent_torsions for " .
      "writing";
  print TMP join("\n", sort keys %f_to_par_map) . "\n";


  return;
}

# This function does exactly what it says, and also uses our happy
# global data to do so.  The maps we must use are in @atom_map_list,
# and %bond_map_list.
sub map_types_to_children {

  my $i;

  # Before we continue, we need to reverse the types hash on the
  # absorbed parent.
  my %ptypesmap;
  foreach ( keys( %{$abs_parent{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$abs_parent{types}{$_}};
  foreach (@atomlist) {
    $ptypesmap{$_} = $type;
  }
  }


  # We want the bond mappings to overwrite the atom mappings, so we'll
  # start by looping through @atom_map_list;
```

```
for ($i = 0; $i <= $#atom_map_list; $i++) {
my $unabsorbed_patom = $i;
my $absorbed_patom = $parent_molecule{tosister}{$i};

# It may happen that we end up with a hydrogen, that doesn't exist
# in the absorbed atom.  We will simply skip over the next
# iteration if this is that case.
unless (defined($absorbed_patom)) {
  next;
}

my $unabsorbed_fatom = ${atom_map_list[$i]}{fragatom};
my $fatom_directory = ${atom_map_list[$i]}{directory};

my $absorbed_fatom =
  $children{$fatom_directory}{tosister}{$unabsorbed_fatom};

# Again, bail out if we're not defined here.
unless (defined($absorbed_fatom)) {
  next;
}

# With that preamble out of the way, find the type we're
# interested in.
my $ptype = $ptypesmap{$absorbed_patom};

# And map it onto the fragment.  Because of the way types is
# stored, we'll need to reverse that hash, store the value, then
# re-reverse it.
my %atypesmap;
foreach ( keys( %{$abs_children{$fatom_directory}{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$abs_children{$fatom_directory}{types}{$_}};
  foreach (@atomlist) {
  $atypesmap{$_} = $type;
  }
}

# Do the re-map.
$atypesmap{$absorbed_fatom} = $ptype;

# Now re-build the re-reversed hash, so we can store it properly.
my %tmp_hash;
foreach (keys %atypesmap) {
  push(@{$tmp_hash{$atypesmap{$_}}}, $_);
}

# And attach it.
$abs_children{$fatom_directory}{types} = { %tmp_hash }
}

# The 'easy' part is done.  We now need to provide a similar mapping
# for the bonds.  We do this second since we're much more interested
# in having the bonds match absolutely correctly.
foreach ( keys %bond_map_list ) {
my $key = $_;
my ( $unabsorbed_patom1, $unabsorbed_patom2 ) =
  split("-", $key);

my %map_hash = %{$bond_map_list{$key}};

my $absorbed_patom1 = $parent_molecule{tosister}{$unabsorbed_patom1};
my $absorbed_patom2 = $parent_molecule{tosister}{$unabsorbed_patom2};

my ($unabsorbed_fatom1, $unabsorbed_fatom2) =
  split("-", $map_hash{fragbond});
my $fatom_directory = $map_hash{directory};

my $absorbed_fatom1 =
  $children{$fatom_directory}{tosister}{$unabsorbed_fatom1};
my $absorbed_fatom2 =
  $children{$fatom_directory}{tosister}{$unabsorbed_fatom2};

# Once again, in the original file, we provided a map from _every_
# bond to fragments in the database.  If there is no map for the
# given atom, we need to simply move on.
unless (defined($absorbed_patom1) and defined($absorbed_patom2) ) {
  next;
}

# With that preamble out of the way, find the types we're
# interested in.
my $ptype1 = $ptypesmap{$absorbed_patom1};
my $ptype2 = $ptypesmap{$absorbed_patom2};

# And map them onto the fragment.  Because of the way types is
# stored, we'll need to reverse that hash, store the value, then
# re-reverse it.
my %atypesmap;
foreach ( keys( %{$abs_children{$fatom_directory}{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$abs_children{$fatom_directory}{types}{$_}};
  foreach (@atomlist) {
  $atypesmap{$_} = $type;
  }
}

# Do the re-map.
$atypesmap{$absorbed_fatom1} = $ptype1;
$atypesmap{$absorbed_fatom2} = $ptype2;

# Now re-build the re-reversed hash, so we can store it properly.
my %tmp_hash;
foreach (keys %atypesmap) {
  push(@{$tmp_hash{$atypesmap{$_}}}, $_);
}

# And attach it.
$abs_children{$fatom_directory}{types} = { %tmp_hash }
}

# That's it, we've mapped all of the types from the parent onto the
# children.  Whether we've done it absolutely correctly remains to
# be tested, in production.

  return;
}

# Once again, we have a function that manipulates global data.  This
# one checks all of the bonds in the %bond_map_list, and determines
# which bond in the child was supposed to represent it.  It then sets
# the bond length in the parent to the bond length of the child.  It
# does not change the coordinates, it simply manipulates the
# information in the mff key of the absorbed parent to match the 'new'
# bond length.  This is somewhat expidited in that we can use the
# unabsorbed space to do this transformation
sub map_fragbond_to_parent {

  # Since we'll be digging out types from the parent, based on atom
  # numbers, we'll neet the reversed hash.
  my %ptypesmap;
  foreach ( keys( %{$abs_parent{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$abs_parent{types}{$_}};
  foreach (@atomlist) {
    $ptypesmap{$_} = $type;
  }
  }

  # Ok, design change.  I took a _huge_ performance hit when I updated
  # the mff each time through.  We'll only update it once, instead,
  # though we will search through it several times.
  my @mfflist = split("\n", $abs_parent{mff});


  foreach (keys %bond_map_list ) {
  my $key = $_;
  my ($patom1, $patom2) = split("-", $_);
  my ($fatom1, $fatom2) = split("-", $bond_map_list{$key}{fragbond});
  my $fdir = $bond_map_list{$key}{directory};

  # Before we go any further, make certain that the parent atoms
  # exist in the absorbed fragment, since we're uninterested in any
  # others.
  unless (exists($parent_molecule{tosister}{$patom1}) and
        exists($parent_molecule{tosister}{$patom2})) {
    next;
  }

  my %frag = %{$children{$fdir}};
  my @atom1_coordinates = @{${$frag{coordinates}}[$fatom1]};
  my @atom2_coordinates = @{${$frag{coordinates}}[$fatom2]};
  my @difference = v_sub(@atom1_coordinates, @atom2_coordinates);
  my $length = v_scalar_len(@difference);

  my $aatom1 = $parent_molecule{tosister}{$patom1};
  my $aatom2 = $parent_molecule{tosister}{$patom2};

  # We know that the atom types will be in alphabetical order.
  # Further, we don't care anymore what that order is, since the
  # actual bond length doesn't depend on it.
  my ($aatom1type, $aatom2type)  =
    sort ($ptypesmap{$aatom1}, $ptypesmap{$aatom2});

  # Now, go through mfflist and mangle away.  We'll simply be
  # changing $_ if it is appropriate.  Note that the following
  # pattern matching is very restrictive for performance (it matches
  # the beginning and the end of the string), and will not change
  # lines that have been commented.
  foreach (@mfflist) {
    my $pattern = "$aatom1type $aatom2type " . '(\d+) (\d+\.\d+)';
    if ($_ =~ /^$pattern$/) {
    my $oldspring = $1;
    my $oldlength = $2;
    # For reference, on the tested system, the biggest length
    # difference was 0.004.
    $_ = "$aatom1type $aatom2type $oldspring $length";
    # print "Need to mangle line, length difference was " .
    # abs($length - $oldlength) . "\n";
    }

  }
  }

  # Now put the mff back where it belongs.
  $abs_parent{mff} = join("\n", @mfflist);

  return;
}


# Once again, we have a function that manipulates global data.  This
# one checks all of the bonds in the @atom_map_list, and deterimines
# which atom in the child was supposed to represent it.  It then sets
# all of the angles centered on that atom in the fragment to the
# approprate angles in the parent.  It does not change the
# coordinates, it simply manipulates the information in the mff key of
# the absorbed parent to match the 'new' bond angles.  This is
# somewhat expedited in that we can use the unablsorbed bits to do
# this transformation.  Note that this task is a bit trickier, since
# there is no simple way to determine which atoms in the fragment
# (besides the central atom) should represent the parent ones.  We'll
# end up relying on qcode similarities for this comparison.

# The following gets us ready to use the CFUNCS package, which is
# simply our own interface to c functions that should not be
# duplicated.  In this case, it's necessary for atom to atom
# comparison.  This will cost us a performance hit, but that's
# acceptable in this context.

use lib "$RealBin/../shlib/cpmodule";
use CFUNCS;

sub map_fragangles_to_parent {

  my $i;

  # Since we'll be digging out types from the parent, based on atom
```

```
# numbers, we'll neet the reversed hash.
my %ptypesmap;
foreach ( keys( %{$abs_parent{types}} ) ) {
my $type = $_;
my (@atomlist) = @{$abs_parent{types}{$_}};
foreach (@atomlist) {
  $ptypesmap{$_} = $type;
}
}

# Reverse the parent's qcode hash and map the atom numbers from the
# tosister hash.  Delete any that are unnecessary.
my %abs_qcodes;
foreach ( keys %{$parent_molecule{qcodes} } ) {
my $qcode = $_;
my @atomlist = @{$parent_molecule{qcodes}{$_}};
foreach (@atomlist) {
  if (exists($parent_molecule{tosister}{$_})) {
  $abs_qcodes{$parent_molecule{tosister}{$_}} = $qcode;
  }
}
}

my @mfflist = split("\n", $abs_parent{mff});

# my $maxdiff = 0;

for ( $i = 0; $i <= $#atom_map_list; $i++ ) {
my $cpatom = $i;   # $cpatom is a pneumonic for central parent atom
my $cfatom = $atom_map_list[$i]{fragatom};
my $fdir = $atom_map_list[$i]{directory};

# Once again, before we go too far, make sure this atom shows up
# in the absorbed parent.
unless (exists($parent_molecule{tosister}{$cpatom})) {
  next;
}

# Ok, for sanity, and to minimize the number of angles we need to
# search for, we'll move to the absorbed atoms / fragments space.
my $acpatom = $parent_molecule{tosister}{$cpatom};
my $acfatom = $children{$fdir}{tosister}{$cfatom};

# Now get lists of neighbors for the parent and fragment
# molecules, both of course, still in absorbed space.
my @parent_neighbors;
my @fragment_neighbors;

my @connectivity = @{$abs_parent{connectivity}{$acpatom}};
foreach (@connectivity) {
  push (@parent_neighbors,$_->[0] );
}

@connectivity = @{$abs_children{$fdir}{connectivity}{$acfatom}};
foreach (@connectivity) {
  push (@fragment_neighbors,$_->[0] );
}

# Ok, this requires a little bit of a trick, and back reference.
# I've already created %abs_qcodes.  We'll do something similar
# here with this fragment.
my %frag_qcodes;
foreach ( keys %{$children{$fdir}{qcodes} } ) {
  my $qcode = $_;
  my @atomlist = @{$children{$fdir}{qcodes}{$_}};
  foreach (@atomlist) {
  if (exists($children{$fdir}{tosister}{$_})) {
    $frag_qcodes{$children{$fdir}{tosister}{$_}} = $qcode;
  }
  }
}

# Before we can pass the qcodes to get_qcode_deviance(), we need
# them to be references to lists, instead of simply text strings.
# We will create 'mirror hashes' of @parent_neighbors and
# @fragment_neighbors for this purpose.
my %parent_neighbors;
my %fragment_neighbors;

foreach (@parent_neighbors) {
  my @qcodes = split(" ", $abs_qcodes{$_});
  $parent_neighbors{$_} = [ @qcodes ];
}

# And do the same for the current fragment.
foreach (@fragment_neighbors) {
  my @qcodes = split(" ", $frag_qcodes{$_});
  $fragment_neighbors{$_} = [ @qcodes ];
}

# Now we need to match up the atoms.  There are n ^ 2 comparisons
# to make, one for each cross reference of parent neighbors to
# fragment neighbors.  In this process, build a %par_to_frag hash.
# We'll check the hash in a second step to make sure no values are
# repeated, if they were, we will not have any other good
# criterion to go on.
my %par_to_frag;
my %frag_to_par;
foreach (@parent_neighbors) {
  my $patom = $_;
  my $best_match = 0;
  my $best_match_frag_atom = -1;
  foreach (@fragment_neighbors) {
  my $fatom = $_;
  if (exists($frag_to_par{$_})) {
    next;
  }
  my $match = CFUNCS::get_qcode_deviance(
                      $parent_neighbors{$patom},
                      $fragment_neighbors{$fatom}
                     );
  # Ok, this isn't a simple problem.  When matches are very
  # close, we really need to simply 'guess' at the proper atoms,
  # and it's absolutely imperative that we don't duplicate
  # atoms, or we won't know how to map backwards.  If the qcodes
  # are 'very close', it really shouldn't make much of a
  # difference.  In this sense, this algorithm is a 'first come,
  # first serve' one, in that the earlier atoms in
  # @parent_neighbors get precedence over the later ones.
  if ($match > $best_match) {
    $best_match = $match;
    $best_match_frag_atom = $fatom;
  }
  }
  $par_to_frag{$patom} = $best_match_frag_atom;
  $frag_to_par{$best_match_frag_atom} = $patom;
}

# Ok!  We now know which children correspond to which parents,
# simply find all of the possible children angles, and map them to
# all of the corresponding parent angles.

my ($i, $j);
my @central_coordinates = @{$abs_children{$fdir}{coordinates}[$acfatom]};
for ($i = 0; $i < $#fragment_neighbors; $i++) {
  my $fatom1 = $fragment_neighbors[$i];
  my $patom1 = $frag_to_par{$fatom1};
  for ($j = $i + 1; $j <= $#fragment_neighbors; $j++) {
  my $fatom2 = $fragment_neighbors[$j];
  my $patom2 = $frag_to_par{$fatom2};

  # Find the relevant angle, it's probably faster to recalculate
  # it than to look it up in the attached .mff file.
  my @atom1_coordinates = @{$abs_children{$fdir}{coordinates}[$fatom1]};
  my @atom2_coordinates = @{$abs_children{$fdir}{coordinates}[$fatom2]};
  my @vec1 = v_sub(@atom1_coordinates, @central_coordinates);
  my @vec2 = v_sub(@atom2_coordinates, @central_coordinates);
  my $angle = acos(v_dot_prod(@vec1, @vec2) /
           (v_scalar_len(@vec1) * v_scalar_len(@vec2))
          ) * 180 / $PI;

  # We've got our angle, just find out what the types are (in
  # the parent), alphabetize them, then look for that line to
  # replace.
  my ($aptype1, $aptype2) =
    ( sort $ptypesmap{$patom1}, $ptypesmap{$patom2} );

  my $aptype_c = $ptypesmap{$acpatom};

  foreach (@mfflist) {
    my $pattern = "$aptype1 $aptype_c $aptype2 " . '(\d+) (\d+\.\d+)';

    if ($_ =~ /^$pattern$/) {
    my $oldspring = $1;
    my $oldangle = $2;
    # For reference, on the tested system, the biggest angle
    # difference was 10 degrees.  This was on a particularly
    # troublesome anisole, which was only optimized at AM1, so
    # it seems reasonable.
    $_ = "$aptype1 $aptype_c $aptype2 $oldspring $angle";
    # my $diff = abs($angle - $oldangle);
    # if ($diff > $maxdiff) {
    #   $maxdiff = $diff
    # }
    # print "Need to mangle line, angle difference was
    # $diff\n";
    }
  }
  }
}
}

# print "The maximum angle difference was $maxdiff\n";

# Now put the mff back where it belongs.
$abs_parent{mff} = join("\n", @mfflist);

return;
}

# This subroutine attaches a torsions key to every member of
# %abs_children, so these values can be manipulated when we actually
# generate the final directories for each torsion.  This will actually
# be a reference to a list of hashes, with each member looking as
# follows:
# $tors{fatomtypes} = "C1 C2 C3 C4" or something like that.  These are
#                     types of the atoms that are driven in the
#                     indicated angle_vs_energy key.
# $tors{patomtypes} = "C5 C2 C3 C6" or so.  These are the types of the
#                     atoms that these fragment atoms correspond to.
# $tors{fatomnums} = "1 2 3 4" Same as fatomtypes, but with the atom
#                              numbers.
# $tors{patomnums} = "12 23 3 19" Same as patomtypes, but with the atom
#                                 numbers.
# $tors{angle_vs_energy} <-- This is the complete angle_vs_energy file
#                            for the given torsion.  If in the
#                            database, the driven atoms are all
#                            present on the absorbed fragment, then
#                            this is simply the Energies file (from
#                            the appropriate torsion directory)
#                            exactly.  If they are not all present,
#                            pick one torsion, and regenerate it from
#                            the .log files.

sub configure_torsions {

  # Once again, since we're looking up types from atom indeces,
  # reverse the parents types hash, and all of the abs_children's
  # types hashes.

  my %ptypesmap;  # Reversed types hash for parent molecule
  foreach ( keys( %{$abs_parent{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$abs_parent{types}{$_}};
  foreach (@atomlist) {
    $ptypesmap{$_} = $type;
  }
  }
```

```
  my %ftypesmap;
  foreach (keys %abs_children ) {
  my $dir = $_;
  foreach ( keys( %{$abs_children{$dir}{types}} ) ) {
    my $type = $_;
    my (@atomlist) = @{$abs_children{$dir}{types}{$_}};
    foreach (@atomlist) {
    $ftypesmap{$dir}{$_} = $type;
    }
  }
  }

  # We'll also need to reverse the parent and children's qcode hashes,
  # so we can search the qcode by index.
  my %pqcode;
  foreach ( keys %{$parent_molecule{qcodes}} ) {
  my $qcode = $_;
  my @atomlist = @{$parent_molecule{qcodes}{$_}};
  foreach (@atomlist) {
    $pqcode{$_} = $qcode;
  }
  }

  my %fqcode;
  foreach (keys %abs_children ) {
  my $dir = $_;
  foreach ( keys %{$children{$dir}{qcodes}} ) {
    my $qcode = $_;
    my @atomlist = @{$children{$dir}{qcodes}{$_}};
    foreach (@atomlist) {
    $fqcode{$dir}{$_} = $qcode;
    }
  }
  }

print '.';

  # Now, we loop through all of the bonds in the parent molecule.  If
  # the bond doesn't exist on the absorbed parent, abandon the
  # interation.  This iteration will only generate a hash of fragment
  # torsions we need, we will use that hash later to actually make the
  # requested entries.
  my %frag_torsions;
  foreach (keys %bond_map_list ) {
  my ($patom1, $patom2) = split ("-", $_);
  my ($fatom1, $fatom2) = split ("-", $bond_map_list{$_}{fragbond});
  my $fdir = $bond_map_list{$_}{directory};

  # Before we get to far, be ready to abandon this iteration
  unless (exists($parent_molecule{tosister}{$patom1}) and
        exists($parent_molecule{tosister}{$patom2})) {
    next;
  }

  my $apatom1 = $parent_molecule{tosister}{$patom1};
  my $apatom2 = $parent_molecule{tosister}{$patom2};
  my $afatom1 = $children{$fdir}{tosister}{$fatom1};
  my $afatom2 = $children{$fdir}{tosister}{$fatom2};

  # The next step is to determine if this bond represents a valid
  # torsion.  If not, again, we can abandon this iteration.
  my @conn1 = @{$abs_parent{connectivity}{$apatom1} };
  my @conn2 = @{$abs_parent{connectivity}{$apatom2} };

  unless (scalar(@conn1) >= 2 and scalar(@conn2) >=2) {
    next;
  }

  # Also, if the bond between the two atoms is aromatic (bond order
  # 1.5), we will not include this torsion.  We'll simply check the
  # existing connectivity hash for the one that points back.
  my $include_torsion = 1;
  foreach (@conn1) {
    my $this_atom = $_->[0];
    unless ($this_atom == $apatom2) {
    next;
    }
    my $bond_order = $_->[1];
    if ($bond_order == 1.5) {
    $include_torsion = 0;
    }
  }

  unless ($include_torsion) {
    next;
  }

######################################################################
## We need to create a section here than handles the fact that some of
## the torsions are not in the database (as they should be).  We'll
## provide a list of 'acceptable' missing torsions.  NOTE THAT THIS
## SECTION IS PURELY TEMPORARY.  REMOVE IT OR REMOVE THE VALUES FROM
## ACCEPTABLY_MISSING WHEN THE TORSIONS ARE COMPLETE
######################################################################

#  my %acceptably_missing = (
#               'C12H18O-1/torsion_15-18'   => 1,
#               'C16H15NO5-0/torsion_16-23' => 1,
#               'C16H18O-0/torsion_3-21'    => 1,
#               'C20H16O3-0/torsion_13-20'  => 1,
#               'C8H18-0/torsion_4-7'       => 1,
#               'C8H18-0/torsion_7-10'      => 1,
#               'C8H18O-0/torsion_13-16'    => 1,
#              );

#  if (exists($acceptably_missing{"$fdir/torsion_${fatom1}-${fatom2}"})) {
    #print "$fdir/torsion_${fatom1}-${fatom2} found but acceptable, ".
    #  "Not recording it as a torsion to calculate\n";
#    next;
#  }

  unless ( -e "$db_path/$fdir/torsion_${fatom1}-${fatom2}/angle_vs_energy") {
  die "Necessary file: " .
    "$db_path/$fdir/torsion_${fatom1}-${fatom2}/angle_vs_energy" .
    " does not exist, cannot continue!\n";
  }

  # Ok, now construct the fragment torsions hash.  The keys of the
  # hash will be the _absorbed_ parent atom bond, and the values
  # will be in the form of a space separated list that looks like:
  # <full_path_to_torsion_directory> <absorbed_fragment_bond>"
  $frag_torsions{"$apatom1 $apatom2"} =
    "$db_path/$fdir/torsion_${fatom1}-${fatom2} $afatom1-$afatom2";
  }

  # We will create a reversed version of this hash so we only need to
  # do one thing per directory.  The reversed version won't have the
  # parent bond in it, instead, we will make the value be the absorbed
  # fragment bond we are calculating.
  my %torsion_dirs;
  foreach ( keys %frag_torsions ) {
  my ($dir, $fragbond) = split(" ", $frag_torsions{$_});
  if (exists($torsion_dirs{$dir}) and $torsion_dirs{$dir} ne $fragbond) {
    die "This is quite suspicious\n";
  } else {
    $torsion_dirs{$dir} = $fragbond;
  }
  }

  foreach (sort keys %torsion_dirs ) {
  my $dir = "$_";
  unless (-e $dir) {
    die "FATAL!  $dir does not exists\n";
    print "Abs_parent bond $torsion_dirs{$_}\n";
  }
  }

print '.';

  # Sweet!  Now we can finally get to the business of creating our
  # list of torsion values to be using.  This is a loop over all the
  # %torsion_dirs.
  foreach (keys %torsion_dirs ) {
  my $torsion_dir = $_;
  my ($afatom1, $afatom2) = split("-", $torsion_dirs{$_});

  my @tmplist = split('/', $torsion_dir);
  my $fdir = $tmplist[-2];

  # We need to determine what torsion (in the unabsorbed fragment) was
  # used to drive the torsion.  We then need also to determine if
  # the two 'end atoms' exist in the absorbed fragment.  If they do,
  # it's very simple for us to provide the angle_vs_energy
  # information, if not, we have to do some mashing.
  my ($ufatom1, $ufatom2) =
    ($abs_children{$fdir}{tosister}{$afatom1},
     $abs_children{$fdir}{tosister}{$afatom2}) ;

  # Additionally, before we move on, we want to initialize the
  # variables $upatom1, $upatom2, $apatom1, and $apatom2.  This will
  # require working back through the %bond_map_list;
  my ($upatom1, $upatom2) = (undef, undef);
  foreach (keys %bond_map_list ) {
    my $pbond = $_;
    my $fbond = $bond_map_list{$_}{fragbond};
    my $thisdir = $bond_map_list{$_}{directory};
    if ($fbond eq "$ufatom1-$ufatom2" and $thisdir eq $fdir) {
    ($upatom1, $upatom2) = split("-", $pbond);
    last;
    }
  }

  unless ( defined($upatom1) and defined($upatom2) ) {
    die "Unable to find parent bond for child bond " .
      "$fdir:$ufatom1-$ufatom2.  Unknown logic error";
  }

  my ($apatom1, $apatom2) = ($parent_molecule{tosister}{$upatom1},
               $parent_molecule{tosister}{$upatom2});

  unless ( defined($apatom1) and defined($apatom2) ) {
    die "Unable to find parent bond in absorbed parent for " .
      "unabsorbed parent bond $fdir:$upatom1-$upatom2.  " .
      "Unknown logic error";
  }

  # Ok, this bond is known for all 4 entities now.

  my @conn1 = @{$children{$fdir}{connectivity}{$ufatom1} };
  my @conn2 = @{$children{$fdir}{connectivity}{$ufatom2} };

  # The algorithm we use for selecting the torsion to drive in
  # torsion_driver.pl is to simply drive the lowest numbered atoms
  # on each end of the bond.  Let's find those.
  my $tufatom1 = 1000000000;  # For 'terminal unabsorbed fragment
                              # atom 1.  This algorithm will not
                              # work for fragments with larger than
                              # 1 billion atoms.
  my $tufatom2 = 1000000000;
  foreach (@conn1) {
    if ($_->[0] < $tufatom1 and $_->[0] != $ufatom2) {
    $tufatom1 = $_->[0];
    }
  }

  foreach (@conn2) {
    if ($_->[0] < $tufatom2 and $_->[0] != $ufatom1) {
    $tufatom2 = $_->[0];
    }
  }

  my %tors = ();     # This is the hash that will be added to the
                       # appropriate child.  It is declared here since
                       # each loop through the torsions will result in
                       # exactly one reference being pushed onto the
                       # %absorbed_children hash.
```

```
if (exists($children{$fdir}{tosister}{$tufatom1}) and
  exists($children{$fdir}{tosister}{$tufatom2})) {
  # If this is true, we can simply use this particular dihedral.
  # Initialize our hash, read the angle_vs_energy file, and get on
  # with it.

  my @aftorsion = ($children{$fdir}{tosister}{$tufatom1},
          $afatom1, $afatom2,
          $children{$fdir}{tosister}{$tufatom2}
         );
  my @uftorsion = ($tufatom1, $ufatom1, $ufatom2, $tufatom2);

  # Now we need to find the 'best matches' for the terminal atoms
  # in the absorbed and unabsorbed parents.  These will be
  # searched only on the parent connectivity.
  my ($tupatom1, $tupatom2, $tapatom1, $tapatom2);

  my @qcode = split(" ", $fqcode{$fdir}{$tufatom1});

  @conn1 = @{$parent_molecule{connectivity}{$upatom1} };
  @conn2 = @{$parent_molecule{connectivity}{$upatom2} };

  my $best_match = 0;
  foreach (@conn1) {
  my $patom = $_->[0];
  if ($patom == $upatom2 ) {
    next;
  }
  my @pqcode = split(" ", $pqcode{$patom});

  # We're ready to get our match.
  my $match = CFUNCS::get_qcode_deviance( \@qcode, \@pqcode );

  if ($match > $best_match) {
    $best_match = $match;
    $tupatom1 = $patom;
  }
  }

  # And do it for the other side
  $best_match = 0;
  @qcode = split(" ", $fqcode{$fdir}{$tufatom2});
  foreach (@conn2) {
  my $patom = $_->[0];
  if ($patom == $upatom1 ) {
    next;
  }
  my @pqcode = split(" ", $pqcode{$patom});

  # We're ready to get our match.
  my $match = CFUNCS::get_qcode_deviance( \@qcode, \@pqcode );

  if ($match > $best_match) {
    $best_match = $match;
    $tupatom2 = $patom;
  }
  }

  my @uptorsion = ($tupatom1, $upatom1, $upatom2, $tupatom2);
  my @aptorsion = ($parent_molecule{tosister}{$tupatom1},
          $apatom1, $apatom2,
          $parent_molecule{tosister}{$tupatom2} );

  # And get the values into our hash.  Remember, the values for
  # the types (and the atom indeces) are for the absorbed parent
  # and fragment, since that's the only domain we're working in
  # for Matt's force field.
  $tors{fatomtypes} = "$ftypesmap{$fdir}{$aftorsion[0]} " .
                  "$ftypesmap{$fdir}{$aftorsion[1]} " .
                  "$ftypesmap{$fdir}{$aftorsion[2]} " .
                  "$ftypesmap{$fdir}{$aftorsion[3]}";

  $tors{patomtypes} = "$ptypesmap{$aptorsion[0]} " .
                  "$ptypesmap{$aptorsion[1]} " .
                  "$ptypesmap{$aptorsion[2]} " .
                  "$ptypesmap{$aptorsion[3]}";

  $tors{fatomnums} = "$aftorsion[0] $aftorsion[1] " .
                 "$aftorsion[2] $aftorsion[3]";

  $tors{patomnums} = "$aptorsion[0] $aptorsion[1] " .
                 "$aptorsion[2] $aptorsion[3]";

  # Now, we can read the angle_vs_energy file, and complete our
  # hash entries.
  open(AVE, "<$torsion_dir/angle_vs_energy") or next and
  die "Unable to open $torsion_dir/angle_vs_energy for reading";
  my @ave = <AVE>;
  close AVE;
  chomp @ave;
  foreach (@ave) {
  $_ =~ s/=>/ /;
  }

  $tors{angle_vs_energy} = join("\n", @ave) . "\n";

} else {

  # We already have the central bond for all 4 entities, pick
  # terminal atoms (lowest numbered ones) from the absorbed
  # fragment
  my @conn1 = @{$abs_children{$fdir}{connectivity}{$afatom1} };
  my @conn2 = @{$abs_children{$fdir}{connectivity}{$afatom2} };

  # The algorithm we use for selecting the torsion to drive in
  # torsion_driver.pl is to simply drive the lowest numbered atoms
  # on each end of the bond.  Let's find those.
  my $tafatom1 = 1000000000;  # For 'terminal absorbed fragment
                              # atom 1.  This algorithm will not
                              # work for fragments with larger than
                                # 1 billion atoms.
  my $tafatom2 = 1000000000;
  foreach (@conn1) {
  if ($_->[0] < $tafatom1 and $_->[0] != $afatom2) {
    $tafatom1 = $_->[0];
  }
  }

  foreach (@conn2) {
  if ($_->[0] < $tafatom2 and $_->[0] != $afatom1) {
    $tafatom2 = $_->[0];
  }
  }

  my($tufatom1, $tufatom2) = ($abs_children{$fdir}{tosister}{$tafatom1},
              $abs_children{$fdir}{tosister}{$tafatom2});

  my (@aftorsion) = ($tafatom1, $afatom1, $afatom2, $tafatom2);
  my (@uftorsion) = ($tufatom1, $ufatom1, $ufatom2, $tufatom2);

  # Once again, we need to rely on qcodes to pick the parent one
  # for us.
  my ($tupatom1, $tupatom2, $tapatom1, $tapatom2);

  my @qcode = split(" ", $fqcode{$fdir}{$tufatom1});

  @conn1 = @{$parent_molecule{connectivity}{$upatom1} };
  @conn2 = @{$parent_molecule{connectivity}{$upatom2} };

  my $best_match = 0;
  foreach (@conn1) {
  my $patom = $_->[0];
  if ($patom == $upatom2 ) {
    next;
  }
  my @pqcode = split(" ", $pqcode{$patom});

  # We're ready to get our match.
  my $match = CFUNCS::get_qcode_deviance( \@qcode, \@pqcode );

  if ($match > $best_match) {
    $best_match = $match;
    $tupatom1 = $patom;
  }
  }

  # And do it for the other side
  $best_match = 0;
  @qcode = split(" ", $fqcode{$fdir}{$tufatom2});
  foreach (@conn2) {
  my $patom = $_->[0];
  if ($patom == $upatom1 ) {
    next;
  }
  my @pqcode = split(" ", $pqcode{$patom});

  # We're ready to get our match.
  my $match = CFUNCS::get_qcode_deviance( \@qcode, \@pqcode );

  if ($match > $best_match) {
    $best_match = $match;
    $tupatom2 = $patom;
  }
  }

  my @uptorsion = ($tupatom1, $upatom1, $upatom2, $tupatom2);
  my @aptorsion = ($parent_molecule{tosister}{$tupatom1},
          $apatom1, $apatom2,
          $parent_molecule{tosister}{$tupatom2} );

# Diagnostics
#foreach ( @aftorsion, @uftorsion, @aptorsion, @uptorsion) {
#  $_++;
#}
#print "In dir $fdir\n";
#print "Unabsorbed parent: " . join(" ", @uptorsion) . "\n";
#print "Absorbed parent: " . join(" ", @aptorsion) . "\n";
#print "Unabsorbed fragment: " . join(" ", @uftorsion) . "\n";
#print "Absorbed fragment: " . join(" ", @aftorsion) . "\n";
#foreach ( @aftorsion, @uftorsion, @aptorsion, @uptorsion) {
#  $_--;
#}

  # The torsions look good, we can now us @uftorsion, look through
  # the log files, and get our own angle vs energy values.

  my %angnrg;
  my @logfiles =
  glob "$db_path/$fdir/torsion_${ufatom1}-${ufatom2}/*.log";
  foreach (@logfiles) {
  my $filename = $_;

  my $retval = get_last_dihedral_and_energy($filename, @uftorsion);
  unless (defined($retval)) {
    die "Unable to get the final angle and energy from log file " .
      "$filename";
  }
  my ($angle, $energy) = split(" ", $retval);
  $angnrg{$angle} = $energy;
  }

  # Normalize the energies so the lowest energy is 0;
  my $minnrg = undef;
  foreach (keys %angnrg ) {
  my $energy = $angnrg{$_};
  unless (defined($minnrg)) {
    $minnrg = $energy;
    next;
  }
  if ($angnrg{$_} < $minnrg) {
    $minnrg = $energy;
  }
  }

  # And do the normalization
  foreach ( keys(%angnrg)) {
  $angnrg{$_} -= $minnrg;
  }
```

```
    # Now that we have all the information, simply initialize %tors
    # And get the values into our hash.  Remember, the values for
    # the types (and the atom indeces) are for the absorbed parent
    # and fragment, since that's the only domain we're working in
    # for Matt's force field.
    $tors{fatomtypes} = "$ftypesmap{$fdir}{$aftorsion[0]} " .
                        "$ftypesmap{$fdir}{$aftorsion[1]} " .
                        "$ftypesmap{$fdir}{$aftorsion[2]} " .
                        "$ftypesmap{$fdir}{$aftorsion[3]}";

    $tors{patomtypes} = "$ptypesmap{$aptorsion[0]} " .
                        "$ptypesmap{$aptorsion[1]} " .
                        "$ptypesmap{$aptorsion[2]} " .
                        "$ptypesmap{$aptorsion[3]}";

    $tors{fatomnums} = "$aftorsion[0] $aftorsion[1] " .
                       "$aftorsion[2] $aftorsion[3]";

    $tors{patomnums} = "$aptorsion[0] $aptorsion[1] " .
                       "$aptorsion[2] $aptorsion[3]";

    # And finally, the angle_vs_energy bit.
    foreach (sort {$a <=> $b } keys %angnrg ) {
    $tors{angle_vs_energy} .= "$_ $angnrg{$_}\n";
    }
  }

  # Before we move on with this loop, we need to push an anonymous
  # reference to the hash onto the proper fragment.

  push @{$abs_children{$fdir}{torsions}}, {%tors};
  }




# Diagnostics
#  foreach (keys %abs_children ) {
#  print "This directory is $_\n";
#  if (exists $abs_children{$_}{torsions}) {
#    my @list = @{$abs_children{$_}{torsions}};
#    print "The members of my list follow: " . join(", ", @list) . "\n";
#    print "The values within the hashes follow:\n";
#    foreach (@list) {
#    my %thishash = %{$_};
#    foreach (keys %thishash ) {
#      print "In hash:  Key = $_, value = $thishash{$_}\n";
#    }
#    }
#  } else {
#    print "No assigned torsions for this directory.  This is most " .
#        "likely because it was on our forgiveness list\n";
#  }
#  }

  # That's it, our torsions are configured, and ready for output.
  return;

}

# This function takes as arguments a molecule, and two atoms to test.
# It will exit with an error if the atoms are not connected, and will
# do a variety of tests on each side of the bond to determine if the
# torsion should be 1 fold, 2 fold, or 3 fold symmetric.

# LIMITATIONS: This function does not pay attention to
# stereochemistry.  As such, it cannot detect the 2 fold symmetry
# around some meso compound.  Also, the individual tests are a bit odd
# to think about, and should be revisisted at some point.
sub find_max_symmetry(\%$$) {
  my $molref = shift;
  my %mol = %{$molref};
  my $atom1 = shift;
  my $atom2 = shift;


  # Reverse the types map, we'll need it.
  my %typesmap;
  foreach ( keys( %{$mol{types}} ) ) {
  my $type = $_;
  my (@atomlist) = @{$mol{types}{$_}};
  foreach (@atomlist) {
    $typesmap{$_} = $type;
  }
  }


  # What we're after (before we do the tests) is a hash for each end
  # of the torsion, whose keys are the types of the connnected atoms,
  # and whose values are how many atoms with that type are on that
  # end of the bond.
  my (@rawconn1) = @{$mol{connectivity}{$atom1} };
  my (@rawconn2) = @{$mol{connectivity}{$atom2} };

  my %types1;
  my %types2;
  my ($atom1ok, $atom2ok) = (0, 0);
  foreach (@rawconn1) {
  my $thisatom = $_->[0];
  if ($thisatom == $atom2) {
    $atom1ok = 1;
    next;
  }
  unless (exists $types1{$typesmap{$thisatom}}) {
    $types1{$typesmap{$thisatom}} = 1;
  } else {
    $types1{$typesmap{$thisatom}}++;
  }
  }

  foreach (@rawconn2) {
  my $thisatom = $_->[0];
  if ($thisatom == $atom1) {
    $atom2ok = 1;
    next;
  }
  unless (exists $types2{$typesmap{$thisatom}}) {
    $types2{$typesmap{$thisatom}} = 1;
  } else {
    $types2{$typesmap{$thisatom}}++;
  }
  }

  # There was a problem with the original plan.  Assymetric atoms can
  # cause torsions to be non-symmetric with some distance away.  We
  # need to look for assymetric atoms on all of the end atoms as well.
  # We will consider this 'as far as we have to look' for now.  An
  # atom is considered assymetric if it has 3 different types, and is
  # a C_1 type of atom.
  foreach (@rawconn1) {
  my $thisatom = $_->[0];
  if ($typesmap{$thisatom} =~ /C_1/ ) {
    if (scalar($mol{connectivity}{$thisatom}) >= 3) {
    return 1;
    }
  }
  }


  # And make sure we got a valid bond
  unless ($atom1ok and $atom2ok) {
  die "Atom numbers $atom1 and $atom2 do not appear to be connected";
  }

  my ($types1count, $types2count) =
  (scalar(keys %types1), scalar(keys %types2) );

  my ($end1totcount, $end2totcount) = (0, 0);
  foreach (keys %types1) {
  $end1totcount += $types1{$_};
  }
  foreach (keys %types2) {
  $end2totcount += $types2{$_};
  }

  # As a last error checking step, we definately need at least one
  # member on each end, or it's not even a torsion.
  if ($types1count < 1 or $types2count < 1) {
  print "Unable to assign symmetry to a non-driveable torsion.  There " .
        "were no atoms on one of the central torsion atoms\n";
  die;
  }

  # Now that we have all of the information needed to do the symmetry
  # tests, let's get underway.  We will simply do case by case tests,
  # if we fail all of the tests, we'll exit and print the information
  # for the failed assignment.

  # If either end has 1 type of atom, and 3 total atoms, it has 3 fold
  # symmetry
  if ($types1count == 1 and $end1totcount == 3) {
  return 3;
  }
  if ($types2count == 1 and $end2totcount == 3) {
  return 3;
  }

  # There also exists the case where one end of the bond has only 1
  # type of atom, but two atoms total.  In all cases, this will have
  # 2 fold symmetry.  Previous work was not catching this case, and
  # as a result, we ended up with some bad torsions.
  if ( $types1count == 1 && $end1totcount == 2 or
       $types2count == 1 && $end2totcount == 2 ) {
    return 2;
  }

  # It's also relatively easy to spot non-symmetric bonds.  This
  # happens when one end has 2 (or 3) different types.  Note that
  # bonds down the center of a meso compound do have 2 fold symmetry,
  # but we don't evaluate for this special case for now.
  if ($types1count >= 2 or $types2count >= 2) {
  return 1;
  }

  # Having passed the 1 fold symmetry test, it's relatively easy to
  # detect all cases of 2 fold symmetry.  Note that the _only_ case we
  # know of right now is where one end is sp2 hybridized, and the
  # other has only one atom connected.
  if ($types1count == 1 and $end1totcount == 2 and $end2totcount == 1) {
  return 2;
  }

  if ($types2count == 1 and $end2totcount == 2 and $end1totcount == 1) {
  return 2;
  }

  # If there's only one atom on either end of the bond, there is
  # reflection symmetry (as would be expected by a cosine series), but
  # there is no 2 or 3 fold symmetry.
  if ($types1count == 1 and $types2count == 1) { return 1; }

  # If we get through all of the tests, but still haven't returned,
  # output the information we have, and then die (ungracefully).
  print "Unknown case for assigning symmetries encountered in ".
      "find_max_symmetry.  The information I have follows\n";
  print "The first atom has $types1count different types of neighbors " .
  "($end1totcount total neighbors)\n";
  foreach (keys %types1) {
  print "\t$types1{$_} atoms of type $_\n";
  }
  print "\n";
  print "The second atom has $types2count different types of neighbors " .
  "($end2totcount total neighbors)\n";
  foreach (keys %types2) {
  print "\t$types2{$_} atoms of type $_\n";
  }
  print "\n";
```

```
  exit;

  return;
}

#-----------------------------------------------------------------------#
#  Copyright 1998 by DevDaily Interactive, Inc.  All Rights Reserved.  #
#-----------------------------------------------------------------------#

# The following function was adapted from:
# http://www.devdaily.com/perl/edu/articles/pl010005/pl010005.shtml.
sub prompt($\@;$) {

  my $prompt = shift;
  my $ansref = shift;
  my @answers = @{$ansref};
  my $default = shift;
  my $oldflush = $|;
  my $reply;
  my $ok_to_finish = 0;

  $prompt .= ' (' . join('|', @answers,) . ') ';

  if ($default) {
  $prompt .= "[$default] ";
  }

  $| = 1;
  until ( $ok_to_finish ) {
  print $prompt;

  $reply = <STDIN> ;
  chomp $reply;
  if ($reply eq '' and $default) {
    $ok_to_finish = 1;
    $reply = $default;
    last;
  }

  foreach (@answers) {
    if ($reply eq $_) {
    $ok_to_finish = 1;
    last;
    }
  }
  }

  $| = $oldflush;
  return $reply;
}

# The following function simply provides the entire interactive
# portion of the last steps of the fitting.  We need to get a list of
# all torsions that need to be fitted, and allow the user to fit them
# to their little heart's content, until they say they are 'done' with
# the work.  This will be done via a series of text menus.
sub enter_interactive {

  my $am_finished = 0;
  my $fragname = 'none';
  my $torsion_index = 0;
  my @choices;

  print "Entering interactive mode:\n";

  # We'll need a copy of the original master.mff file, as we'll
  # regularly be appending to the existing one, and it's easiest to
  # reconstitute it if we have a copy handy.
  my $mff_original;
  my $oldsep = $/;
  $/ = undef;
  open TMP, "<master/master.mff" or
  die "Unable to open master/master.mff for reading";
  $mff_original = <TMP>;
  close TMP;
  $/ = $oldsep;

  # We will look for a file in the master directory named (aptly
  # enough) 'completed_torsions'.  Each line will have the following:
  # <fragment directory>,<torsion number>,<mff torsion line>.  We can
  # compare against this has to determine if a torsion is finished or
  # not.  We also append these parameters to the end of the .mff file
  # when doing verification.  Finally, we append both these, and the
  # translated versions of them to the .mff file when we say we're
  # 'done'.  In order to prevent adding duplicates to this list later,
  # the values will be references to hashes, whose keys are the name
  # of the torsion, and values are the parameters.
  my %completed_torsions = ();
  if (-e "master/completed_torsions") {
  open TMP, "<master/completed_torsions" or
    die "Unable to open master/completed_torsions for reading";
  while (<TMP>) {
    chomp;
    my ($dir, $num, $tors, $params) = split(",", $_);
    ${$completed_torsions{"$dir,$num"}}{$tors} = $params;
  }

  close TMP;
  }

  # We need to read the frag_to_parent_torsions file in the master
  # directory, and initialize a hash for translating the fitted child
  # torsions onto the parent molecule.  %ftop will be formatted such
  # that the keys will be the torsion spec for the child, the values
  # will be the torsion spec for the parent.  The directory will be
  # ignored, since all atom labels in the entire system will be
  # unique.
  my %ftop;
  open TMP, "<master/frag_to_parent_torsions" or
  die "Unable to open master/frag_to_parent_torsions";
  while (<TMP>) {
  chomp $_;
  my @values = split(" ", $_);
  my $key = join(" ", @values[1..4]);
  my $value = join(" ", @values[5..8]);

  $ftop{$key} = $value;
  }
  close TMP;

  # We need to initialize the hash which will contain all of the
  # torsions we have to do, and what their current status is.  The
  # status can be one of the following: Not fitted, Torsion done,
  # Verify done, or Finished.  A torsion will not be marked finished
  # unless the user declares it so, or it has an entry in the master
  # directory.  After we decide the format on this, make sure we
  # initialize this properly.

  my %torsions;
  my @scratch = glob '*/torsion*.create';
  foreach (@scratch) {
  $_ =~ m|^(.*)/torsion_(\d+)| or
    die "Odd logic error initializing in enter_interactive";

  # If the directories exist, we assume the work has been done, this
  # may or may not end up giving false information.

  if (exists($completed_torsions{"$1,$2"})) {
    $torsions{"$1,$2"} = "Finished";
  } elsif (-e "$1/verify_$2" ) {
    $torsions{"$1,$2"} = "Verify done";
  } elsif (-e "$1/fit_$2") {
    $torsions{"$1,$2"} = "Torsion done";
  } else {
    $torsions{"$1,$2"} = "Not fitted";  # The default.
  }

  # And just so we provide _some_ torsion to start with
  $fragname = $1;
  $torsion_index = $2;
  }


  until ($am_finished) {
  # Initialize our menu's;
  my $main_menu = <<TXT;

1)  Change current fragment/torsion ($fragname, $torsion_index)
2)  Delete all information for current fragment/torsion
3)  List all torsions with their status
4)  Fit current torsion
5)  Check log file from last fitting run
6)  Verify current torsion
7)  View graph of fit
8)  Declare this torsion finished
9)  Quit
Your choice?
TXT
chomp $main_menu;

  @choices = qw /1 2 3 4 5 6 7 8 9/;

  my $task = prompt($main_menu, @choices);

  if ( $task == 1) {

  # Change the current fragment and or torsion
  print "\n";
  my @choicelist = sort keys %torsions;
  my $thisprompt;
  my $i = 1;

  foreach (@choicelist) {
    my ($fragname, $torsnum) = split(',', $_);
    my $key = $_;
    $thisprompt .= "$i)  $fragname, $torsnum ($torsions{$key})\n";
    $i++;
  }

  $thisprompt .= "Which one?";
  my @numbers = (1..scalar(@choicelist));
  my $thischoice = prompt($thisprompt, @numbers);
  ($fragname, $torsion_index) = split(",", $choicelist[$thischoice - 1]);

  } elsif ($task == 2) {

  # Delete the fit_# and verify_# directories.
  print "\nDeleting $fragname/fit_$torsion_index and " .
      "$fragname/verify_$torsion_index\n";
  system "rm -rf $fragname/fit_$torsion_index\n";
  system "rm -rf $fragname/verify_$torsion_index\n";

  # Delete any graph that may have been created
  system "rm graphs/${fragname}*";

  $torsions{"$fragname,$torsion_index"} = "Not Fitted";

  } elsif($task == 3) {

  # Display all torsions with their status
  print "\n";
  foreach (sort keys(%torsions)) {
    my ($fragname, $torsnum) = split(',', $_);
    print "Fragment $fragname, torsion $torsnum is $torsions{$_}\n";
  }

  } elsif ($task == 4) {

  # Run minimize on the current fragment, make the user wait for it
  # to finish.  When it's finished, move all of the information to
  # the appropriate directory, and report some basic information
  chdir 'master';
  my $command =
    "minimize ../$fragname/torsion_${torsion_index}.create > min.j_log";
  print <<MSG;
The program will appear to freeze while the minimization is being done.
Be patient, or check top to make sure it's still running if you are
concerned.
```

```
MSG

  print "Running command: $command\n";

  system("$command");

  # Before moving the files, do a quick check to see that the log
  # file has ended appropriately.
  open TMP, "<min.j_log" or
    die "Unable to open min.j_log, which we presumabely just created";

  {
    # We definately don't need to keep the data around from slurping
    # up the whole file, so we do this within a block
    my (@logfile) = <TMP>;
    close TMP;
    unless ($logfile[-1] =~ /^c_temp\[\d+\]/) {
    print "Warning, it appears the minimization did not complete " .
          "properly, I recommend you take a look at it to make " .
        "sure it's what you want\n";
    }
  }

  # Now move the files into their own directories, unceremoniously
  # overwriting anything that may already be there.
  unless (-e "../$fragname/fit_${torsion_index}") {
    mkdir "../$fragname/fit_${torsion_index}" or
    die "Unable to create directory ../$fragname/fit_${torsion_index}";
  }

  system "mv min.* ../$fragname/fit_${torsion_index}/";

  $torsions{"$fragname,$torsion_index"} = "Torsion done";

  chdir '..';
  print "Fit completed\n";

  } elsif ($task == 5) {

  # This is the very simple task of simply looking at the last log
  # file with a pager.
  my $command ="less $fragname/fit_${torsion_index}/min.j_log";
  system $command;


  } elsif ($task == 6) {

  # This is very similar to $task == 4 case.  The code is mostly
  # copied from that section.  Note, however, that we have an
  # additional task, which is to (temporarily) append the new
  # torsion to the file.  We will do the mapping when we 'finalize'
  # the new mapping.

  # Find the appropriate parameters, and append them to the
  # master.mff file.
  open TMP, "<$fragname/fit_${torsion_index}/min.delta_params" or
    die "Unable to open $fragname/fit_${torsion_index}/min.delta_params" .
      " for reading";
  my @lines = <TMP>;
  close TMP;
  chomp @lines;

  open TMP, ">>master/master.mff" or
    die "Unable to open master/master.mff for appending";
  print TMP "#\n# torsion\n";
  print TMP join("\n", @lines) . "\n";
  print TMP "# end\n#";
  close TMP;

  chdir 'master';
  my $command =
    "minimize ../$fragname/torsion_${torsion_index}.verify > min.j_log";
  print <<MSG;
The program will appear to freeze while the verification is being done.
Be patient, or check top to make sure it's still running if you are
concerned.
MSG

  print "Running command: $command\n";

  system("$command");

  # Now move the files into their own directories, unceremoniously
  # overwriting anything that may already be there.
  unless (-e "../$fragname/verify_${torsion_index}") {
    mkdir "../$fragname/verify_${torsion_index}" or
    die "Unable to create directory ../$fragname/verify_${torsion_index}";
  }

  system "mv min.* ../$fragname/verify_${torsion_index}/";

  $torsions{"$fragname,$torsion_index"} = "Verify done";

  chdir '..';

  # Before finishing, we need to restore the original master.mff
  # file.
  open TMP, ">master/master.mff" or
    die "Unable to open master/master.mff for writing";
  print TMP $mff_original;
  close TMP;

  print "Verification finished\n";

  } elsif ($task == 7) {

  # Here we simply have gnuplot pop up a window with the graph.  It
  # will pause for 10 seconds, then go away.  Note that we'll
  # actually do the offset to the data in the min.fit_pe_delta file,
  # and create a 'temporary' version of this.
  print "Popping up a graph of the requested data.  It will remain " .
        "onscreen until you hit return in the window you see this " .
        "text in\n";

  # The following (commented out) section is for viewing the
  # information in the min.fit_pe_delta directory, and is not indeed
  # a verification.  Perhaps it should go into another main menu
  # option?

#  my %fitpe;
#  open TMP, "<$fragname/verify_${torsion_index}/min.fit_pe_delta" or
#    die "Unable to open $fragname/fit_${torsion_index}/min.fit_pe_delta " .
#      "for reading";
#  while (<TMP>) {
#    my ($angle, $y1, $y2) = split(' ', $_);
#    $fitpe{$angle} = "$y1 $y2";
#  }

  # Find our offset;
#  my $offset = undef;
#  foreach (keys %fitpe) {
#    my $angle = $_;
#    my ($y1, $y2) = split(" ", $fitpe{$angle});
#    unless (defined $offset) {
#    $offset = $y1;
#    }
#    if ($y1 < $offset) {
#    $offset = $y1;
#    }
#    if ($y2 < $offset) {
#    $offset = $y2;
#    }
#  }

  # Apply it to all the values in the hash.
#  foreach (keys %fitpe) {
#    my $angle = $_;
#    my ($y1, $y2) = split(" ", $fitpe{$angle});
#    $y1 -= $offset;
#    $y2 -= $offset;
#    $fitpe{$angle} = "$y1 $y2";
#  }

  # And print them to a temp file;
#  open TMP, ">data.tmp" or
#    die "Unable to open data.tmp for writing";
#  foreach (sort {$a <=> $b} keys(%fitpe) ) {
#    print TMP "$_ $fitpe{$_}\n";
#  }
#  close TMP;

  # Now create the input file for gnuplot.  Note that in order for
  # the backslashes to make it to the input file, we need to escape them.
#  open TMP, ">gnuplot_parms.txt" or
#    die "Unable to open gnuplot_parms.txt for writing";
#  print TMP <<INST;
#set xrange [0:360]
#set data style points
#set xtics 30
#plot \\
#  "data.tmp" using 1:2 title "Dataset 1" with points ps 1, \\
#  "data.tmp" using 1:3 title "Dataset 2" with points ps 1, \\
#  "$fragname/angle_vs_energy_${torsion_index}" title "Original Data" \\
#    with points ps 1
#pause 10
#INST
#  close TMP;

  # If we want to save this to a file, we'll need to play with set
  # output and set terminal, but it should be pretty easy.

  # After a bit of research, this is extraordinarily easy.  Add a
  # line:
  # set postscript eps
  # Before the plot command, then simply run:
  # system "gnuplot blah > output.eps" to get the plot

  # Before we can generate the graph (in eps form), we need to gather
  # a bit of information
  unless (-e "./graphs") {
  mkdir "./graphs" or
    die "Unable to make directory ./graphs";
  }

  # Get our driven torsion.
  open TMP, "<$fragname/torsion_${torsion_index}.create" or
  die "Unable to open $fragname/torsion_${torsion_index}.create " .
      "for reading.";
  while ( <TMP> ) {
  if ( $_ =~ /\#  n_min_interv/ ) {
    last;
  }
  }

  my $line = <TMP>;
  chomp $line;
  close TMP;

  my ($tatom1, $tatom2, $tatom3, $tatom4, undef) = split " ", $line;

  $tatom1++;
  $tatom2++;
  $tatom3++;
  $tatom4++;

  # We have our torsion, now get our parameters.
  open TMP, "<$fragname/fit_${torsion_index}/min.delta_params" or
  die "Unable to open fragname/fit_${torsion_index}/min.delta_params " .
      "for reading";
  my $dparams = <TMP>;
  close TMP;
  chomp $dparams;
  my (undef, undef, undef, undef, undef, @plist) = split " ", $dparams;

  $dparams = join(" ", @plist);

  # Before we try to plot things, we need to get the min_params info into a
  # 0 based format, and document the offset.
```

```
  my $coffset = undef;
  open TMP, "<$fragname/verify_${torsion_index}/min.pe_total" or
    die "Unable to open $fragname/verify_${torsion_index}/min.pe_total " .
        "for reading";
  my %cangnrg;
  while ( <TMP> ) {
    chomp;
    my ($angle, $energy) = split;
    unless ( defined ($coffset) ) {
      $coffset = $energy;
    }
    if ( $coffset > $energy ) {
      $coffset = $energy;
    }
    $cangnrg{$angle} = $energy;
  }
  close TMP;

  # Now, do the offset
  foreach ( keys %cangnrg ) {
    $cangnrg{$_} -= $coffset;
  }

  # And print it to our own temp file
  open TMP, ">offset_classical.txt" or
    die "Unable to open offset_classical.txt for writing";
  foreach ( sort { $a <=> $b } keys %cangnrg ) {
    print TMP "$_ $cangnrg{$_}\n";
  }
  close TMP;

  # We've got what we need, now just build our gnuplot input file
  open TMP, ">./gnuplot_eps.txt" or
  die "Unable to open ./gnuplot_eps.txt for writing";
  print TMP <<TXT;
set terminal postscript eps
set xrange [0:360]
set yrange [0:]
set data style points
set xtics 30
set title "${fragname}_$tatom1-$tatom2-$tatom3-
$tatom4\\n\\n$dparams\\n\\nClassical energy offset:  $coffset" ,12
"TimesNewRomanPSMT,24"
plot \\
  "$fragname/angle_vs_energy_${torsion_index}" \\
    title "Ab intio energies" with points ps 1, \\
  "offset_classical.txt" \\
    title "Classical energies" with lines
TXT
  close TMP;

  system "gnuplot gnuplot_eps.txt > " .
  "graphs/${fragname}_$tatom1-$tatom2-$tatom3-$tatom4.eps";

  # And get rid of the file.
  unlink "./gnuplot_eps.txt" or
  die "Unable to delete ./gnuplot_eps.txt";

  # Here's the one we pop up.
  open TMP, ">gnuplot_parms.txt" or
    die "Unable to open gnuplot_parms.txt for writing";
  print TMP <<INST;
set xrange [0:360]
set yrange [0:]
set data style points
set xtics 30
set title "${fragname}_$tatom1-$tatom2-$tatom3-
$tatom4\\n$dparams\\n\\nClassical energy offset:  $coffset"
"TimesNewRomanPSMT,24"
plot \\
  "$fragname/angle_vs_energy_${torsion_index}" \\
    title "Ab intio energies" with points ps 1, \\
  "offset_classical.txt" \\
    title "Classical energies" with lines
pause -1
INST
  close TMP;

  # Now pop up the graph for our viewers.
  system "gnuplot gnuplot_parms.txt";

  # And finally, clean up after ourselves.
  unlink "gnuplot_parms.txt" or
    die "Unable to unlink gnuplot_parms.txt";
  unlink "offset_classical.txt" or
    die "Unable to unlink offset_classical.txt";

  # Done with viewing graph section

  } elsif ($task == 8) {

  # When a torsion is 'declared finished', several things happen.
  # First, we add it to the %completed_torsions hash.  Then, we
  # write the information into the completed_torsions file.
  # Finally, we add an appropriate section to $mff_original, and
  # finally, we over-write master.mff.  Note that if it's already
  # finished, we won't do anything, but inform the user of that.

  if  ($torsions{"$fragname,$torsion_index"} eq "Finished" ) {
    print "Torsion $fragname,$torsion_index is already finished!\n";
    next;
  };


  open TMP, "<$fragname/fit_${torsion_index}/min.delta_params" or
    die "Unable to open $fragname/fit_${torsion_index}/min.delta_params" .
      " for reading";
  my @params = <TMP>;
  close TMP;
  chomp @params;

  # Look through the params, and add them to %completed_torsions as
  # appropriate
  foreach (@params) {
    my @scratch = split(" ", $_);
    my $tors = join(" ", @scratch[0..3]);
    my $params = join(" ", @scratch[4..$#scratch]);
    ${$completed_torsions{"$fragname,$torsion_index"}}{$tors} = $params;
  }

  # And write the whole hash into the file.  We want to avoid (for
  # cleanliness, at least) duplicating what torsions are done.
  open TMP, ">master/completed_torsions" or
    die "Unable to open master/completed_torsions for writing";
  foreach (sort keys %completed_torsions) {
    my $key = $_;
    my ($dir, $number) = split(",", $_);
    my %theselines = %{$completed_torsions{$key}};
    foreach (sort keys %theselines) {
    my $tors = $_;
    my $params = $theselines{$tors};
    # Now we can print out an apprpriate line
    print TMP "$dir,$number,$tors,$params\n";
    }
  }
  close TMP;

  # Okies, this next section will be a bit funky.  What we're going
  # to do is loop over all of the params.  For each one, we make
  # sure that there is a value in %ftop, if there's not, it's a
  # critical error.  We'll then extract the parameters from the
  # current iteration, create a new line with the value of the
  # current %ftop entry, and splice it into the list.

  my $i;
  for ($i = 0; $i < scalar(@params); $i++) {
    my $fragline = $params[$i];
    my @scratch = split(" ", $fragline);
    my $child_torsion = join(" ", @scratch[0..3]);
    my $params = join(" ", @scratch[4..$#scratch]);
    my $parent_torsion = $ftop{$child_torsion};
    unless (defined $parent_torsion) {
    die "Was unable to find an appropriate parent torsion for this " .
        "child torsion\n";
    }

    # We're set, do the splice, and increment $i, since we're
    # sticking something in the list.
    splice(@params, $i + 1, 0, "$parent_torsion $params");
    $i++;
  }

  # Now that we have all of the params to add to the master force
  # field, do it to $mff_original, then write it out.  That's the
  # end.
  $mff_original .= <<INP;
#
# Mapped torsions from fragment $fragname, torsion $torsion_index
#
# torsion
INP
  $mff_original .= join("\n", @params) . "\n";
  $mff_original .= <<INP;
# end
#
INP

  # And update our 'declared finished' master force field.
  open TMP, ">master/master.mff" or
    die "Unable to open master/master.mff for writing";
  print TMP $mff_original;
  close TMP;

  # Finally, let's mark this guy as finished;
  $torsions{"$fragname,$torsion_index"} = "Finished";

  } elsif ($task == 9) {
  # This is simply the quit option.
  $am_finished = 1;
  } else {
  die "Unknown option $task encountered";
  }


  }

  print "Exiting interactive mode\n";
  return;
}


# The following function searches through all of the directories, and
# runs fits an verifies on all of the torsions, it then tries to run
# them all, and finally, it exits
sub fit_all_torsions {

  chdir 'myff' or
  die "Unable to change to myff directory\n";

  # Before we go crazy here, let's give the user some options.

  my $menu = <<TXT;
Note that even after running the torsions, you will need to manually
check them to make sure the fits are good, etc.  Feel free to simply
re-run this program after the batch is done, then select s (to skip
the directory initialization).  Also, be certain to enter the new
values into the master force field.
a)  Run all possible fits and verifies
u)  Run all unfinished (as marked in the master/completed_torsions
    file)
o)  Run only torsions for which there is no fit or verify directory
q)  Quit
Your choice?
TXT
chomp $menu;

  my @choices = qw /a u o q/;
```

```
my $task = prompt($menu, @choices);

my %completed_torsions = ();
if (-e "master/completed_torsions") {
open TMP, "<master/completed_torsions" or
  die "Unable to open master/completed_torsions for reading";
while (<TMP>) {
  chomp;
  my ($dir, $num, $tors, $params) = split(",", $_);
  ${$completed_torsions{"$dir,$num"}}{$tors} = $params;
}

close TMP;
}

# We need to initialize the hash which will contain all of the
# torsions we have to do, and what their current status is.  The
# status can be one of the following: ( Note that this nomenclature
# is a bit different than in the interactive mode ).  It will simply
# be a string of characters, v means the verify is done, f means the
# fit is done, c means it has been completed.

my %torsions;
my @scratch = glob '*/torsion*.create';
foreach (@scratch) {
$_ =~ m|^(.*)/torsion_(\d+)| or
  die "Odd logic error initializing in enter_interactive";

# Make each of the entries an empty string, so we can concatcanate
# to it.
$torsions{"$1,$2"} = '';

# If the directories exist, we assume the work has been done, this
# may or may not end up giving false information.

if (exists($completed_torsions{"$1,$2"})) {
  $torsions{"$1,$2"} .= 'c';
}
if (-e "$1/fit_$2") {
  $torsions{"$1,$2"} .= 'f';
}
if (-e "$1/verify_$2" ) {
  $torsions{"$1,$2"} .= 'v';
}
}

# Before we start with the tasks, we need to create a backup of the
# master.mff file, so we can append the appropriate parameters to it
# before doing the verify steps.
system 'cp master/master.mff master/master.mff.bak';

if ($task eq 'a') {
# In this case, we delete all old information, and run all jobs

my @tasklist;
foreach (keys %torsions ) {
  my ($dir, $index) = split(",", $_);
  push @tasklist,
  "minimize ../$dir/torsion_$index.create > min.j_log";
  push @tasklist, "mkdir ../$dir/fit_$index";
  push @tasklist, "mv min.* ../$dir/fit_$index/";
  push @tasklist, "echo '# torsion' >> master.mff;" .
                 "cat ../$dir/fit_$index/min.delta_params " .
          ">> master.mff;" .
          "echo '# end' >> master.mff";
  push @tasklist,
  "minimize ../$dir/torsion_$index.verify > min.j_log";
  push @tasklist, "mkdir ../$dir/verify_$index";
  push @tasklist, "mv min.* ../$dir/verify_$index/";
  push @tasklist, "cp master.mff.bak master.mff";
}

print "User requested " . scalar(@tasklist) . " tasks\n";
my $i;

system 'rm -rf */fit_*';
system 'rm -rf */verify_*';

chdir 'master' or die "Unable to change directory to master";

for ($i = 0; $i < @tasklist; $i++) {
  my $job = $tasklist[$i];
  print "Running: $job (" . ($i + 1) . "/" .
  scalar(@tasklist) . ")\n";
  system "$job";
}

chdir '..' or die "Unable to change back to parent directory";

} elsif ($task eq 'u') {

# In this case, we only run jobs that aren't marked as finished,
# according to the finished torsions file.
my @tasklist;
foreach (keys %torsions ) {
  my ($dir, $index) = split(",", $_);

  unless ($torsions{$_} =~ /c/) {

  push @tasklist,
    "minimize ../$dir/torsion_$index.create > min.j_log";
  push @tasklist, "mkdir ../$dir/fit_$index";
  push @tasklist, "mv min.* ../$dir/fit_$index/";
  push @tasklist, "echo '# torsion' >> master.mff;" .
                  "cat ../$dir/fit_$index/min.delta_params " .
           ">> master.mff;" .
           "echo '# end' >> master.mff";
  push @tasklist,
    "minimize ../$dir/torsion_$index.verify > min.j_log";
  push @tasklist, "mkdir ../$dir/verify_$index";
  push @tasklist, "mv min.* ../$dir/verify_$index/";
  push @tasklist, "cp master.mff.bak master.mff";
  }
}

print "User requested " . scalar(@tasklist) . " tasks\n";
my $i;

chdir 'master' or die "Unable to change directory to master";

for ($i = 0; $i < @tasklist; $i++) {
  my $job = $tasklist[$i];
  print "Running: $job (" . ($i + 1) . "/" .
  scalar(@tasklist) . ")\n";
  system "$job";
}

chdir '..' or die "Unable to change back to parent directory";

} elsif ($task eq 'o') {

# In this case, we only run the corresponding verify or fit jobs
# if the directory doesn't already exist
my @tasklist;
foreach (keys %torsions ) {
  my ($dir, $index) = split(",", $_);

  unless ($torsions{$_} =~ /f/) {

  push @tasklist,
    "minimize ../$dir/torsion_$index.create > min.j_log";
  push @tasklist, "mkdir ../$dir/fit_$index";
  push @tasklist, "mv min.* ../$dir/fit_$index/";
  }

  unless ($torsions{$_} =~ /v/) {
  push @tasklist, "echo '# torsion' >> master.mff;" .
                  "cat ../$dir/fit_$index/min.delta_params " .
           ">> master.mff;" .
           "echo '# end' >> master.mff";
  push @tasklist,
    "minimize ../$dir/torsion_$index.verify > min.j_log";
  push @tasklist, "mkdir ../$dir/verify_$index";
  push @tasklist, "mv min.* ../$dir/verify_$index/";
  push @tasklist, "cp master.mff.bak master.mff";
  }
}

print "User requested " . scalar(@tasklist) . " tasks\n";
my $i;

chdir 'master' or die "Unable to change directory to master";

for ($i = 0; $i < @tasklist; $i++) {
  my $job = $tasklist[$i];
  print "Running: $job (" . ($i + 1) . "/" .
  scalar(@tasklist) . ")\n";
  system "$job";
}

chdir '..' or die "Unable to change back to parent directory";


} elsif ($task eq 'q') {
system "rm master/master.mff.bak";
exit;
} else {
die "Unknown command $task in batch mode processing";
}



exit;
}
```

general

Atom Handling Library

atom.h

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */
```

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

/* Conditional includes */
#ifndef VECTOR_H
#define VECTOR_H
#include "vector.h"
#endif

/* The following is a memory debugging library */
#ifdef DMALLOC
#include <dmalloc.h>
#endif

/* Defines */

/* The following constant was taken from: Mills, I., Cvitas, T.,
   Homann, K., Kallay, N., and Kuchitsu, K. "Quantities, Units and
   Symbols in Physical Chemistry"  The Green Book, 2nd Edition,
   Blackwell Sci., 1993.   The error in the last significant digits (1
   standard deviation) is 36.  */
#define AVAG_NUMBER 6.0221367e23

/* The following defines are convenience constants, for using */
/* atom_list_manage() */
#define A_CLEAR 0
#define A_COUNT 1
#define A_PUSH 2
#define A_POP 3
#define A_SHIFT 4
#define A_UNSHIFT 5

/* The following defines are convenience constants, for using */
/* recurse_molecule_do() */
#define RMD_SAVESTATES 1
#define RMD_NOTOUCHLIST 2
#define RMD_CLEANMEM 4
#define RMD_NOINITSTATES 8

#ifndef QDEPTH
#define QDEPTH 20
#endif

/* Typedefs and structs */
#ifndef BOOLEAN
#define BOOLEAN
typedef enum {false = 0, no = 0, true = 1, yes = 1} boolean;
#endif

#ifndef MAXSTR
#define MAXSTR 256
#endif

/* This inline macro will speed up getting interatomic distances.
   Since it's a macro, it cannot do any type checking, be careful with
   it.  Note that this is definately _not_ the fastest algorithm for
   finding the distance.  Using a Taylor series to some arbitrary
   precision would be faster in almost all cases.  Regardless, until
   we have a need to speed up the simulation, we will use this macro
   to get distances.  (The key to speeding this up is to avoid the
   usage of the sqrt function)  */
#define get_interatomic_distance(atom1, atom2)                        \
        sqrt( ( (atom1)->coordinates[0] - (atom2)->coordinates[0] ) * \
              ( (atom1)->coordinates[0] - (atom2)->coordinates[0] ) + \
              ( (atom1)->coordinates[1] - (atom2)->coordinates[1] ) * \
              ( (atom1)->coordinates[1] - (atom2)->coordinates[1] ) + \
              ( (atom1)->coordinates[2] - (atom2)->coordinates[2] ) * \
              ( (atom1)->coordinates[2] - (atom2)->coordinates[2] ) )

/* This is the first attempt (9-7-00) to unify the atom strucure type */
typedef struct atom {
  char *label;
  unsigned int atomic_number;
  vector_d coordinates;

  /* Note:  it is implicit that the pointers to in the *bond array */
  /* are filled in numerical order.  The bonds will range from 0 to */
  /* (valence - 1) */
  int valence;
  struct atom **bond;
  float *bond_order;
  float formal_charge;
  long double *qcode;
  double charge;

  /* This added 7-03-01 to handle asymmetric carbons */
  char s_descriptor; /* For 'stereochemical' descriptor.  The original
           implementation will use rules for prioritizing
           based on qcodes.  This is not the standard CIP
           descriptors we're used to, but the method will
           give a unique identifier regardless, as long
           as the differences in the groups surrounding
           the carbon in question are no further than
           QDEPTH - 1 atoms away */

  /* Note also, that for the original implementation, I will be using
  'r' and 's' (instead of the capitalized versions) for my own
  purposes.  Finally, '\0' should be the default value */

  /* The other pointer can be used to record any information that may */
  /* need to be attached to the atom in question.  Note that it is very */
  /* important to manually assign typing to the void pointers, since */
  /* it will happily read and write in very bad ways. */
  void **other;

  /* The state int can be used for many purposes.  It is particularly */
  /* useful for recording how many times an atom has been accessed, */
  /* and is also instrumental to recursive access */
  long int state;

  struct atom *next;
  struct atom *previous;
} atom;
```

```
/* Functions in atom_handling.c, by type */

/* Output functions */

void print_atom(FILE *out, atom *atom_to_print);
void print_molecule(FILE *out, atom* molecule_to_print);
void print_molecule_connectivity(FILE *out, atom *some_atom);
void atom_print_xyz(FILE *out_stream, atom *molecule_member);
void atom_print_com(FILE *out_stream, atom *molecule_member);

/* Input functions */

atom *read_init_formatted_coordinates(FILE *read_stream);
void read_formatted_connectivity(FILE *read_stream, atom *member);

/* Functions that primarily allocate/deallocate memory */

atom *atom_realloc(atom *base_atom_address, int new_array_size,
        int lab_length);
atom *duplicate_molecule(atom *old_molecule);
atom *repack_molecule(atom *old_molecule);
void free_molecule(atom *molecule_member);

/* Functions that generate lists */
int *generate_connectivity_list(atom *some_atom);
int *generate_angle_list(atom *some_atom);

/* Boolean query functions, that provide information on a particular atom */

boolean is_atom_valence_full(atom *some_atom);
boolean is_connected(atom *atom1, atom* atom2);
boolean is_bonded(atom *atom1, atom* atom2);
boolean is_asymmetric_carbon(atom *some_atom);
int i_is_asymmetric_carbon(atom *some_atom);
boolean does_it_have_s_descriptor(atom *some_atom);
int i_does_it_have_s_descriptor(atom *some_atom);
boolean is_methyl_group(atom *some_atom);
boolean is_aromatic(atom *some_atom);
atom *initialize_blank_atom(void);

/* Bond mashing functions, this includes manipulation of connectivity,
   and querying of these properties */

void atom_connect(atom *atom1, atom *atom2, float bond_order);
void atom_disconnect(atom *atom1, atom *atom2);
int atom_get_number_of_bonds(atom *this_atom);
float get_atom_to_atom_bond_order(atom *atom1, atom *atom2);
float get_stored_bond_order(atom *atom1, atom *atom2);
float get_single_bond_length(int atomic_number1, int atomic_number2);
void atom_pack_this_atom_bonds(atom* some_atom);
void molecule_pack_all_bonds(atom *molecule);
float get_total_bond_order(atom *this_atom);
boolean verify_molecule_connectivity(atom *some_atom);
void repair_connectivity(atom *some_atom);

/* Recursive access to a molecule, and support functions for this
   access (mostly state mashing ) */

int recurse_molecule_do(int (*some_function)(atom *),
           atom *this_atom, int depth, int mode);
int recurse_molecule_do_core( int (*do_function)(atom *),
              atom *this_atom, int depth, int mode);
void molecule_zero_states(atom *some_atom);
void molecule_max_states(atom *some_atom);
long int *save_states(atom *some_atom);
int restore_states(atom *some_atom, long int *old_states);

/* Functions for setting/querying stereochemical information on atoms */

void assign_q_s_descriptor(atom *some_atom);
atom *gimme_higher_priority(atom *atom1, atom *atom2);

/* Information for querying/setting other properties of atoms */

int atom_lab_to_num(char *label);
int assign_formal_charge(atom *this_atom);
float atom_get_pauling_elecneg(int atomic_number);
void molecule_normalize_charges(atom *member, float total_charge);
/* See the macro: double get_interatomic_distance(atom1, atom2) */

/* Querying information on, and moving around full molecules */

int molecule_get_size(atom *molecule);
int get_atom_offset(atom *this_atom);
atom *molecule_return_base(atom *some_atom);

/* Modification of the geometry/connectivity of a molcule */

void delete_molecule_group( atom *tokeep, int bond_number );

/* Atom list management (this may always have only one function? */
atom *atom_list_manage(atom *some_atom, int op_code);

/* Miscelleneous functions */
int i_noop_a (atom *);


/* bit packing and reading.  These should be phased out (if they're
   not already), since it's much easier to simply use bitwise
   operators.  I know I started the phase out, but am not sure I've
   gotten rid of it in all places.  */

void mark2(long int *number, int);
void unmark2(long int *number, int);
boolean check2(long int value, int place);

/* The following function resides in ../log2str/get_bond_order.c, but */
/* is prototyped here as the atom_handling.c collection uses it */
float get_bond_order(int atomic_num1, int atomic_num2, float distance);

/* The following functions resides in ../qdb/qdb_shared_functions.c, but */
/* is protyped here as it's used in atom_handling.c.  Note that this */
/* object file (qdb_shared_functions.o) _must_ be linked here whenever this */
```

```
/* file is compiled into a program */
void warn_out(char *message);
void error_exit(char *message);
```

atom_handling.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "atom.h"

/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* The following function is purely diagnostic.  It displayes (in a semi-
   nice way) the values contained in a single atom */

void print_atom(FILE *out, atom *toprint) {

  int i;

  if (toprint == NULL) {
  warn_out("NULL pointer passed to print_atom(), returning without "
       "printing");
  return;
  }

  fprintf(out, "This atom's base address: %x\n", (unsigned int)toprint);
  fprintf(out, "This atom's offset(1-base):%d\n", get_atom_offset(toprint) +
1);
  fprintf(out, "Atom label:              \"%s\"\n", toprint->label);
  fprintf(out, "Atomic Number:           %u\n", toprint->atomic_number);
  fprintf(out, "Coordinates:             %f, %f, %f\n", toprint-
>coordinates[0],
      toprint->coordinates[1], toprint->coordinates[2]);
  fprintf(out, "Valence:                 %d\n", toprint->valence);
  fprintf(out, "Bonds (pointers):        ");
  for (i = 0; i < toprint->valence; i++) {
  if ( i != 0 ) { fprintf(out, ", "); }
  fprintf(out, "%x", (unsigned int)toprint->bond[i]);
  }
  fprintf(out, "\n");
  fprintf(out, "Bonds (base 1 offset):   ");
  for (i = 0; i < toprint->valence; i++) {
  if ( i != 0 ) { fprintf(out, ", "); }
  fprintf(out, "%d", get_atom_offset(toprint->bond[i]) + 1);
  }
  fprintf(out, "\n");
  fprintf(out, "Bonds_orders:            ");
  for (i = 0; i < toprint->valence; i++) {
  if ( i != 0 ) { fprintf(out, ", "); }
  fprintf(out, "%g", toprint->bond_order[i]);
  }
  fprintf(out, "\n");
  fprintf(out, "Formal charge:           %g\n", toprint->formal_charge);
  fprintf(out, "Stereochemical Descriptor:%c\n", toprint->s_descriptor);
  fprintf(out, "qcode[0] (pointer):      %x\n", (unsigned int)toprint->qcode);
  fprintf(out, "Charge:                  %f\n", toprint->charge);
  fprintf(out, "Other (pointer):         %x\n", (unsigned int)toprint->other);
  fprintf(out, "State:                   %ld\n", toprint->state);
  fprintf(out, "Next atom (pointer):     %x\n", (unsigned int)toprint->next);
  fprintf(out, "Previous atom (pointer): %x\n", (unsigned int)toprint-
>previous);


  return;
}

/* This function simply calls print_atom() for each atom */
/* in the molecule passed to it */
void print_molecule(FILE *out_file, atom *molecule) {

  int i, size;

  if (molecule == NULL) {
  warn_out("NULL pointer passed to print_molecule()");
  }

  molecule -= get_atom_offset(molecule);
  size = molecule_get_size(molecule);

  for (i = 0; i < size; i++) {
  print_atom(out_file, molecule + i);
  fprintf(out_file, "\n");
  }

  return;
}

void print_molecule_connectivity(FILE *out, atom *some_atom) {
  atom *molecule_base;
  int *connectivity, i = 0;

  if (!some_atom) {
  warn_out("NULL atom passed to print_molecule_connectivity()");
  }

  molecule_base = some_atom - get_atom_offset(some_atom);

  connectivity = generate_connectivity_list(molecule_base);

  while( !(connectivity[i] == 0 && connectivity[i + 1] == 0)) {

  fprintf(out, "%d %d %g\n", connectivity[i], connectivity[i + 1],
     get_stored_bond_order(&molecule_base[connectivity[i]],
                &molecule_base[connectivity[i + 1]]));
  i+=2;
  }

  /* Free the connectivity list */
  free(connectivity);

  return;

}

/* The following function gets the posted bond order between atom1,
   and atom2.  Additionally, it verifies that the bond orders are equal
   (from both directions */
float get_stored_bond_order(atom *atom1, atom *atom2) {

  int i;
  float bond_order;
  boolean is_ok = no;

  if ( atom1 == NULL || atom2 == NULL ) {
  error_exit("NULL pointer passed to get_stored_bond_order()");
  }

  for(i = 0; i < atom1->valence; i++) {
  if ( atom1->bond[i] == atom2 ) {
    is_ok = yes;
    bond_order = atom1->bond_order[i];
  }
  }

  if ( !is_ok ) {
  warn_out("Atom1 passed to get_stored_bond_order() is not connected"
       "to atom2");
  }

  is_ok = no;

  for(i = 0; i < atom2->valence; i++) {
  if ( atom2->bond[i] == atom1 ) {
    if ( atom2->bond_order[i] != bond_order ) {
    error_exit("Atoms pass to get_stored_bond_order() have different "
        "bond orders on both ends.  This means the molecule "
        "has been incorrectly initialized, and is fatal");
    }
    is_ok = yes;
  }
  }

  if ( !is_ok ) {
  warn_out("Atom2 passed to get_stored_bond_order() is not connected "
      "to atom1");
  }

  return bond_order;
}

/* This function returns a connectivity list to the calling
   environment for the molecule containing the atom provided.  The
   list is a pairwise list of connected atoms.  Note that this
   function allocates memory, and in order to be clean, it must later
   be free'd. */
int *generate_connectivity_list(atom* some_atom) {

  int *connectivity = NULL, connectivity_index;
  atom *work_atom, *other_end, *molecule_base;
  int i, j;
  long int *old_states;

  /* Since this function should only be used in initializations (not
   dynamically, in any kind of simulation), it will not be
   particularly efficient in its use of realloc() */

  if (!some_atom) {
  warn_out("NULL atom passed to generate_connectivity_list()");
  }

  /* Initialize connectivity */
  if ( ( connectivity = malloc( 2 * sizeof(int) ) ) == NULL ) {
  error_exit("Unable to allocate initial space for the connectivity "
       "list in generate_connectivity_list()");
  }
  connectivity[0] = 0;
  connectivity[1] = 0;
  connectivity_index = 0;

  molecule_base = molecule_return_base(some_atom);
  old_states = save_states(molecule_base);

  molecule_zero_states(molecule_base);

  /* Note that the following algorithm was developed before I had a
   reasonable understanding of bitwise operators.  It should
   _definately_ be re-written at some point */

  for (work_atom = molecule_base; work_atom; work_atom = work_atom->next ) {
  for (i = 0; i < work_atom->valence; i++ ) {
    if (!check2(work_atom->state, i)) {
```

```c
    /* Mark this bond (on both sides) */
    mark2( &(work_atom->state), i );
    other_end = work_atom->bond[i];
    /* And find the remote bond that points back here */
    for ( j = 0; j < other_end->valence; j++ ) {
      if ( other_end->bond[j] == work_atom ) {
      mark2( &(other_end->state), j );
      break;
      }
    }

    /* If we're here, we have 2 more numbers to add. */
    if( ( connectivity =
        realloc(connectivity,
            (connectivity_index + 2) * 2 * sizeof(int))) == NULL ){
      error_exit("Unable to reallocate memory for connectivity "
           "list in generate_connectivity_list()");
    }

    connectivity[connectivity_index * 2] = get_atom_offset(work_atom);
    connectivity[connectivity_index * 2 + 1] = get_atom_offset(other_end);
    connectivity[connectivity_index * 2 + 2] = 0;
    connectivity[connectivity_index * 2 + 3] = 0;

    connectivity_index++;
    }
  }
  }

  restore_states(molecule_base, old_states);

  return connectivity;
}

/* The next function initializes (or re-allocates) space for a new atom, */
/* at the end of the array base passed in the first argument.  It also */
/* initializes all of it's members that are pointers.  It sets all */
/* pointers to NULL, and all values to 0 before it returns a pointer to */
/* the new structure */
atom *atom_realloc(atom *base_p, int new_array_size, int lab_length) {

  atom *new_atom;
  int last_element;
  size_t tot_size = new_array_size * sizeof(atom);
  size_t lab_size = lab_length * sizeof(char);
  /* get the number of the last element of the array from the information */
  /* passed in the argument list */
  last_element = new_array_size - 1;

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  if ( ( base_p = realloc(base_p, tot_size)) == NULL) {
  error_exit("Failed to expand molecule (out of memory) in atom_realloc");
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  new_atom = &base_p[last_element];

  if((new_atom->label = malloc(lab_size * sizeof(char))) == NULL) {
  error_exit(
        "Cannot allocate memory for label of work atom in initialization"
        );
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  /* The following malloc was gravely in error.  It was allocating enough */
  /* space for 3 vector_d's.  On some systems, a pointer to double is the */
  /* same size as a double, but unfortunately, this is not true on all */
  /* systems.  (Perhaps not unfortunately, this was a subtle problem).  */
  /* we need to make coordinates point to enough space for 3 doubles, */
  /* and unfortunately, on some systems, a double is twice the size of */
  /* a pointer to double (or whatever size it might be) */

  if(!( new_atom->coordinates = malloc(3 * VECDVALSIZE) ) ) {
  error_exit(
        "Cannot allocate memory for coordinates of work atom in "
        "initialization"
        );
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  if( ( new_atom->bond_order = malloc(sizeof(float) ) ) == NULL) {
  error_exit(
        "Cannot allocate memory for initial bond_order of work atom in "
        "intialization");
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  /* All space that needed to be allocated are now in place, now we simply */
  /* have the 'hard' initializations to perform */
  new_atom->label[0] = '\0';
  new_atom->atomic_number = 0;

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  new_atom->coordinates[0] = 0.0;
  new_atom->coordinates[1] = 0.0;
  new_atom->coordinates[2] = 0.0;
  new_atom->valence = 0;

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  /* The previous usage of the following line was quite faulty, but in a */
  /* very subtle way.  When we initialize the new bond, we want the bond */
  /* itself to point to null, not the first element of it, the old line */
  /* was "new_atom->bond[0] = NULL;" */
  new_atom->bond = NULL;
  new_atom->bond_order[0] = 0;
  new_atom->formal_charge = 0.0;
  new_atom->qcode = NULL;
  new_atom->charge = 0.0;
  new_atom->s_descriptor = '\0';

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  new_atom->other = NULL;
  new_atom->state = 0;
  new_atom->next = NULL;
  new_atom->previous = NULL;

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  return base_p;
}


/* The following function assigns atomic numbers to the appropriate field */
/* in the atom type it receives, based on the label */
int atom_lab_to_num(char *label) {
  switch (label[0]) {
  case 'A':
  switch (label[1]) {
  case 'c':
    return 89;
  case 'g':
    return 47;
  case 'l':
    return 13;
  case 'm':
    return 95;
  case 'r':
    return 18;
  case 's':
    return 35;
  case 't':
    return 85;
  case 'u':
    return 79;
  default:
    return 0;
  }
  case 'B':
  switch (label[1]) {
  case '\0':
    return 5;
  case 'a':
    return 56;
  case 'e':
    return 4;
  case 'i':
    return 83;
  case 'k':
    return 97;
  case 'r':
    return 35;
  default:
    return 0;
  }
  case 'C':
  switch (label[1]) {
  case '\0':
    return 6;
  case 'a':
    return 20;
  case 'd':
    return 48;
  case 'e':
    return 58;
  case 'f':
    return 98;
  case 'l':
    return 17;
  case 'r':
    return 24;
  case 's':
    return 55;
  case 'u':
    return 29;
  default:
    return 0;
  }
  case 'D':
  switch (label[1]) {
  case 'y':
    return 66;
  default:
    return 0;
  }
  case 'E':
  switch (label[1]) {
  case 'r':
    return 68;
  case 'u':
    return 63;
  default:
    return 0;
```

```
}
case 'F':
switch (label[1]) {
case '\0':
  return 9;
case 'e':
  return 26;
case 'r':
  return 87;
default:
  return 0;
}
case 'G':
switch (label[1]) {
case 'a':
  return 31;
case 'd':
  return 64;
case 'e':
  return 32;
default:
  return 0;
}
case 'H':
switch (label[1]) {
case '\0':
  return 1;
case 'e':
  return 2;
case 'f':
  return 72;
case 'g':
  return 80;
case 'o':
  return 67;
default:
  return 0;
}
case 'I':
switch (label[1]) {
case '\0':
  return 53;
case 'n':
  return 49;
case 'r':
  return 77;
default:
  return 0;
}
case 'K':
switch (label[1]) {
case 'r':
  return 36;
default:
  return 0;
}
case 'L':
switch (label[1]) {
case 'a':
  return 57;
case 'i':
  return 3;
case 'r':
  return 103;
case 'u':
  return 71;
default:
  return 0;
}
case 'M':
switch (label[1]) {
case 'd':
  return 101;
case 'g':
  return 12;
case 'n':
  return 25;
case 'o':
  return 42;
default:
  return 0;
}
case 'N':
switch (label[1]) {
case '\0':
  return 7;
case 'a':
  return 11;
case 'b':
  return 41;
case 'd':
  return 60;
case 'e':
  return 10;
case 'i':
  return 28;
case 'o':
  return 102;
case 'p':
  return 93;
default:
  return 0;
}
case 'O':
switch (label[1]) {
case '\0':
  return 8;
case 's':
  return 76;
default:
  return 0;
}
case 'P':
switch (label[1]) {
case '\0':
  return 15;
case 'd':
  return 46;
case 'o':
  return 84;
case 't':
  return 78;
case 'u':
  return 94;
default:
  return 0;
}
case 'R':
switch (label[1]) {
case 'a':
  return 88;
case 'b':
  return 37;
case 'e':
  return 75;
case 'h':
  return 45;
case 'n':
  return 86;
case 'u':
  return 44;
default:
  return 0;
}
case 'S':
switch (label[1]) {
case '\0':
  return 16;
case 'b':
  return 51;
case 'c':
  return 21;
case 'e':
  return 34;
case 'i':
  return 14;
case 'm':
  return 62;
case 'n':
  return 50;
case 'r':
  return 38;
default:
  return 0;
}
case 'T':
switch (label[1]) {
case 'a':
  return 73;
case 'b':
  return 65;
case 'e':
  return 52;
case 'h':
  return 90;
case 'i':
  return 22;
case 'l':
  return 81;
case 'm':
  return 69;
default:
  return 0;
}
case 'U':
switch (label[1]) {
case '\0':
  return 92;
default:
  return 0;
}
case 'V':
switch (label[1]) {
case '\0':
  return 23;
default:
  return 0;
}
case 'W':
switch (label[1]) {
case '\0':
  return 74;
default:
  return 0;
}
case 'X':
switch (label[1]) {
case 'e':
  return 56;
default:
  return 0;
}
case 'Y':
switch (label[1]) {
case '\0':
  return 39;
case 'b':
  return 70;
default:
  return 0;
}
case 'Z':
switch (label[1]) {
case 'n':
  return 30;
case 'r':
  return 40;
default:
```

```
    return 0;
  }
  default:
  return 0;
  }
  /* End of outside case statement */
  return 0;
}


/* The following function takes as arguments two (struct) atoms, and */
/* connects them along the bonds both ways.  It needs to find the first */
/* non-null space available.  It also reallocates the bond space */
/* to support the new pointer, and adds one to the valence */
/* Note that it will not connect atoms that are already connected */
void atom_connect(atom *atom1, atom *atom2, float bond_order) {

  if ( !atom1 ) {
  error_exit("First atom passed to atom_connect() was NULL");
  }

  if ( !atom2 ) {
  error_exit("Second atom passed to atom_connect() was NULL");
  }

  /* Finally, don't connect them if they're already connected! */
  if (is_connected(atom1, atom2)) {
  return;
  }

  /* First, we expand the valence and reallocate memory for the extra bond */
  atom1->valence++;
  if ((atom1->bond = realloc(atom1->bond, sizeof(atom *) *
             atom1->valence)) == NULL ) {
  error_exit("Failed to expand the valence of atom1 in atom_connect");
  }
  atom2->valence++;
  if ((atom2->bond = realloc(atom2->bond, sizeof(atom *) *
             atom2->valence)) == NULL ) {
  error_exit("Failed to expand the valence of atom2 in atom_connect");
  }

  /* Then, we expand the bond order matrix and likewise reallocate memory */
  if ((atom1->bond_order
     = realloc(atom1->bond_order, sizeof(float) * atom1->valence)) == NULL ) {
  error_exit("Failed to expand the number of bonds of atom1 in atom_connect");
  }
  if ((atom2->bond_order
     = realloc(atom2->bond_order, sizeof(float) * atom2->valence)) == NULL ) {
  error_exit("Failed to expand the number of atoms of atom2 in atom_connect");
  }

  /* Then, we connect them */
  atom1->bond[atom1->valence - 1] = atom2;
  atom2->bond[atom2->valence - 1] = atom1;

  /* And assign the new bond order */
  atom1->bond_order[atom1->valence - 1] = bond_order;
  atom2->bond_order[atom2->valence - 1] = bond_order;

  return;
}

/* The following function is the inverse of atom_connect, and is designed */
/* to allow the user to 'edit' their molecule.  Note that this is a */
/* powerful thing to do, and will mess up the total bond order, formal */
/* charge, and possibly other things */
void atom_disconnect(atom *atom1, atom *atom2) {

  int i;

  if ( !atom1 ) {
  error_exit("First atom passed to atom_disconnect() was NULL");
  }

  if ( !atom2 ) {
  error_exit("Second atom passed to atom_disconnect() was NULL");
  }

  /* Note:  This is much easier to do than atom_connect, since */
  /* atom_pack_this_atom_bonds() does all of the memory work */

  for (i = 0; i < atom1->valence; i++) {
  if (atom1->bond[i] == atom2) {
    atom1->bond[i] = NULL;
  }
  }

  for (i = 0; i < atom2->valence; i++) {
  if (atom2->bond[i] == atom1) {
    atom2->bond[i] = NULL;
  }
  }

  atom_pack_this_atom_bonds(atom1);
  atom_pack_this_atom_bonds(atom2);

  return;
}

/* The following function takes an atomic number as an argument, and */
/* returns the pauling electronegativity.  There are several cases in */
/* this function where one of the return values are commented out. */
/* All of the electronegativities that are returned with 3 significant */
/* figures will be considered correct, and were taken from: */
/* overlap.chem.cornell.edu:8080/~landram/electroneg.sm.html */
float atom_get_pauling_elecneg(int atomic_number) {

  /* Quick note:  case takes an int as it's argument, not a char */
  switch (atomic_number) {
  case 1:
  return 2.20;
  case 2:
  return 0;
  case 3:
  return 0.98;
  case 4:
  return 1.57;
  case 5:
  return 2.04;
  case 6:
  return 2.55;
  case 7:
  return 3.04;
  case 8:
  return 3.44;
  case 9:
  return 3.98;
  case 10:
  return 0;
  case 11:
  return 0.93;
  case 12:
  return 1.31;
  case 13:
  return 1.61;
  case 14:
  return 1.90;
  case 15:
  return 2.19;
  case 16:
  return 2.58;
  case 17:
  return 3.16;
  case 18:
  return 0;
  case 19:
  return 0.82;
  case 20:
  return 1.00;
  case 21:
  return 1.36;
  case 22:
  return 1.54;
  case 23:
  return 1.63;
  case 24:
  return 1.66;
  case 25:
  return 1.55;
  case 26:
  return 1.83;
  case 27:
  return 1.88;
  case 28:
  return 1.91;
  case 29:
  return 1.90;
  case 30:
  return 1.65;
  case 31:
  return 1.81;
  case 32:
  return 2.01;
  case 33:
  return 2.18;
  case 34:
  return 2.55;
  case 35:
  return 2.96;
  case 36:
  return 3.00;
  case 37:
  return 0.82;
  case 38:
  return 0.95;
  case 39:
  return 1.22;
  case 40:
  return 1.33;
  case 41:
  return 1.60;
  case 42:
  return 2.16;
  case 43:
  return 1.90;
  case 44:
  return 2.20;
  case 45:
  return 2.28;
  case 46:
  return 2.20;
  case 47:
  return 1.93;
  case 48:
  return 1.69;
  case 49:
  return 1.78;
  case 50:
  return 1.96;
  case 51:
  return 2.05;
  case 52:
  return 2.10;
  case 53:
  return 2.66;
  case 54:
  return 2.60;
  case 55:
  return 0.79;
  case 56:
  return 0.89;
  case 57:
  return 1.10;
  case 58:
  return 1.1;
  case 59:
  return 1.1;
```

```
    case 60:
    return 1.1;
    case 61:
    return 1.1;
    case 62:
    return 1.2;
    case 63:
    return 1.1;
    case 64:
    return 1.2;
    case 65:
    return 1.1;
    case 66:
    return 1.2;
    case 67:
    return 1.2;
    case 68:
    return 1.2;
    case 69:
    return 1.3;
    case 70:
    return 1.1;
    case 71:
    return 1.3;
    case 72:
    return 1.30;
    case 73:
    return 1.50;
    case 74:
    return 2.36;
    case 75:
    return 1.90;
    case 76:
    return 2.20;
    case 77:
    return 2.20;
    case 78:
    return 2.28;
    case 79:
    return 2.54;
    case 80:
    return 2.00;
    case 81:
    return 2.04;
    case 82:
    return 2.33;
    case 83:
    return 2.02;
    case 84:
    return 2.00;
    case 85:
    return 2.20;
    case 86:
    return 2.4;
    case 87:
    return 0.7;
    case 88:
    return 0.9;
    case 89:
    return 1.1;
    case 90:
    return 1.3;
    case 91:
    return 1.5;
    case 92:
    return 1.4;
    case 93:
    return 1.4;
    case 94:
    return 1.3;
    case 95:
    return 1.3;
    case 96:
    return 1.3;
    case 97:
    return 1.3;
    case 98:
    return 1.3;
    case 99:
    return 1.3;
    case 100:
    return 1.3;
    case 101:
    return 1.3;
    case 102:
    return 1.3;
    } /* End of case statement */
    return 0;
}

/* The following function's purpose should be pretty self evident. Note */
/* that before valence was part of the atom structure, this function was */
/* a bit more complicated */
int atom_get_number_of_bonds(atom *this_atom) {
  return this_atom->valence;
}

/* The following function calls get_bond_order in a friendly way */
float get_atom_to_atom_bond_order(atom *atom1, atom* atom2) {

  /* vector_d work_vector; */
  float bond_order, length, diff1, diff2, diff3;

  /* Get normal values.  This is a long way to write it, but much simpler */
  /* to understand.  A reasonable optimization level will eliminate this */
  /* in the final code */
  /* work_vector = vec_subtract(atom1->coordinates, atom2->coordinates, 3); */
  /* length = vec_scalar_length(work_vector, 3); */

  /* work_vector was allocated by vec_scalar_length, so it */
  /* must be freed */
  /* free(work_vector); */

  /* The previous implementation took _much_ more time than it needed to */
  /* in trial runs.  Instead of using the vector library, we'll just get */
  /* the length ourselves */
  diff1 = atom1->coordinates[0]  - atom2->coordinates[0];
  diff2 = atom1->coordinates[1]  - atom2->coordinates[1];
  diff3 = atom1->coordinates[2]  - atom2->coordinates[2];
  length = sqrt(diff1 * diff1 + diff2 * diff2 + diff3 * diff3);

  bond_order = get_bond_order( atom1->atomic_number, atom2->atomic_number,
               length);

  if ( (atom1->atomic_number == 1 || atom2->atomic_number == 1) &&
     !(atom1->atomic_number == 1 && atom2->atomic_number == 1) &&
     bond_order == -1 && length < 2.0 ) {
  printf("Got trouble\n");
  printf("Atom1 = %d, Atom2 = %d, length = %g\n", get_atom_offset(atom1) + 1,
       get_atom_offset(atom2) + 1, length);
  error_exit("");
  }

  return bond_order;
}


/* The following function checks to see if an atom has all of it's */
/* valence's filled.  It relies quite a bit on the fact that some atoms */
/* have constant valences.  Note that if the function does not have */
/* specific instructions concerning what the valence of that atoms should */
/* be, it always returns true.  Also, note that it subtracts the formal */
/* charge of the molecule before it checks.  This, of course, assumes that */
/* the formal charges have been correctly assigned in the first place */
boolean is_atom_valence_full(atom *some_atom) {

  float total_valence;

  if (some_atom == NULL) {
  warn_out("NULL pointer passed to is_atom_valence_full()");
  }

  total_valence = get_total_bond_order(some_atom) - some_atom->formal_charge;

  switch (some_atom->atomic_number) {
  case 1:
  case 3:
  case 9:
  case 11:
  /* Note that for the halogens, the valence can be larger than one.  This */
  /* is a compromise, and users not doing 'typical' organic chemistry are */
  /* encouraged to not use this function, or to use at their own */
  /* discretion. */
  case 17:
  case 19:
  case 35:
  case 53:
  if (total_valence == 1.0 ) return true;
  else return false;
  case 4:
  case 8:
  case 12:
  if (total_valence == 2.0 ) return true;
  else return false;
  case 5:
  case 7:
  case 13:
  if (total_valence == 3.0 ) return true;
  else return false;
  case 6:
  if (total_valence == 4.0 ) return true;
  else return false;
  default:
  return true;
  }

  /* To keep the compiler happy */
  return true;
}

/* The following function can be used to find out how far an atom is */
/* from the beginning of the linked list.  It is intended to provide useful */
/* information only in the case that the molecule is packed as an array */
int get_atom_offset(atom *this_atom) {

  atom *work_atom;

  /* Error checking */
  if (this_atom == NULL) {
  warn_out("NULL pointer passed to get_atom_offset, this may well be "
       "a fatal error");
  }

  work_atom = molecule_return_base(this_atom);

  return (this_atom - work_atom);
}

/* This function provides a duplicate of another molecule.  It requires */
/* that the molecule be packed in a normal array, as it uses the array */
/* mechanism to access all members to copy.  Later, a function will be */
/* provided to translate molecules in random bond linked, and normal */
/* linked formats to get into this form.  There are a couple of other */
/* details to note:  First, the atom * that is returned will correspond */
/* to the same atom (in the molecule) that *old_molecule pointed to. */
/* Secondly, several parts of the new atom are not copied.  These are: */
/* qcode and state.  The qcodes can be re-established with a call to */
/* assign_qcodes(new_molecule).  The state is zeroed since it's a new */
/* molecule */
atom *duplicate_molecule(atom *old_molecule) {

  int i, j, old_size;
  atom *new_molecule, *work_atom, *old_head;

  /* Find the beginning of the old_molecule */
  old_head = old_molecule - get_atom_offset(old_molecule);

  /* Determine the size of the molecule (how many atoms) */
```

```
  old_size = 1;
  work_atom = old_head;
  for (old_size = 1, work_atom = old_head; work_atom->next;
     old_size++, work_atom = work_atom->next) {}

  if( ( new_molecule = malloc(old_size * sizeof(atom) ) ) == NULL ) {
  error_exit("Failed to malloc memory for new atom in "
         "duplicate_molecule()");
  }

  /* Now, initialize all values in each element of the new_atom_array */
  for (i = 0; i < old_size; i++) {
  if( ( new_molecule[i].label = malloc( ( strlen(old_head[i].label)
                     + 1) * sizeof(char) ) ) == NULL ) {
    error_exit("failed to obtain memory for new label in "
         "duplicate_molecule()");
  }
  if( (new_molecule[i].coordinates = malloc(3 * VECDVALSIZE))==NULL) {
    error_exit(
         "Cannot allocate memory for coordinates of work atom in "
         "duplicate_molecule()");
  }
  if( ( new_molecule[i].bond = malloc(sizeof(atom *) *
                   old_head[i].valence)) == NULL) {
    error_exit( "Cannot allocate memory for initial bond of work atom in "
          "duplicate_molecule()");
  }
  if( ( new_molecule[i].bond_order = malloc(sizeof(float) *
                       old_head[i].valence) ) == NULL) {
    error_exit( "Cannot allocate memory for initial bond_order of "
          "work atom in duplicate_molecule()");
  }

  new_molecule[i].label =
    strcpy(new_molecule[i].label, old_head[i].label);
  new_molecule[i].atomic_number = old_head[i].atomic_number;
  new_molecule[i].coordinates[0] = old_head[i].coordinates[0];
  new_molecule[i].coordinates[1] = old_head[i].coordinates[1];
  new_molecule[i].coordinates[2] = old_head[i].coordinates[2];
  new_molecule[i].valence = old_head[i].valence;
  /* This next bit is a bit tricky.  Make the new bonds correspond */
  /* to the connectivity of the new molecule */
  for (j = 0; j < old_head[i].valence; j++) {
    new_molecule[i].bond[j] =
    new_molecule + get_atom_offset(old_head[i].bond[j]);
    new_molecule[i].bond_order[j] = old_head[i].bond_order[j];
  }
  new_molecule[i].formal_charge = old_head[i].formal_charge;
  new_molecule[i].qcode = NULL;
  new_molecule[i].charge = old_head[i].charge;
  new_molecule[i].s_descriptor = old_head[i].s_descriptor;
  new_molecule[i].other = NULL;
  new_molecule[i].state = 0;
  new_molecule[i].next = NULL;
  new_molecule[i].previous = NULL;
  /* And finally, fix the next/previous links */
  if ( i != 0 ) { new_molecule[i].previous = new_molecule + i - 1; }
  if ( i != (old_size - 1) ) {
    new_molecule[i].next = new_molecule + i + 1;
  }
  }

  return new_molecule + get_atom_offset(old_molecule);
}

/* This function returns the single bond distance (ideal) given two */
/* atomic numbers.  It's initialization data is purely copied from */
/* ../log2str/get_bond_order.c, and as such, should probably be */
/* combined at some point */
float get_single_bond_length(int atomic_number1, int atomic_number2) {

  float single_bond_radius[106]; /* Tables only go up to Iodine */
  int i;

  if (atomic_number1 > 105) {
  fprintf(stderr, "\nBad atom #%d passed to get_bond_order\n",
      atomic_number1);
  return -1.0;
  }
  if (atomic_number2 > 105) {
  fprintf(stderr, "\nBad atom #%d passed to get_bond_order\n",
      atomic_number2);
  return -1.0;
  }

  for (i = 0; i < 106; i++) {single_bond_radius[i] = 0;}

  single_bond_radius[1] = 0.299;
  single_bond_radius[4] = 1.06;
  single_bond_radius[5] = 0.83;
  single_bond_radius[6] = 0.767;
  single_bond_radius[7] = 0.702;
  single_bond_radius[8] = 0.659;
  single_bond_radius[9] = 0.619;
  single_bond_radius[13] = 1.18;
  single_bond_radius[14] = 1.090;
  single_bond_radius[15] = 1.088;
  single_bond_radius[16] = 1.052;
  single_bond_radius[17] = 1.023;
  single_bond_radius[31] = 1.25;
  single_bond_radius[32] = 1.22;
  single_bond_radius[33] = 1.196;
  single_bond_radius[34] = 1.203;
  single_bond_radius[35] = 1.199;
  single_bond_radius[49] = 1.41;
  single_bond_radius[50] = 1.39;
  single_bond_radius[51] = 1.37;
  single_bond_radius[52] = 1.391;
  single_bond_radius[53] = 1.395;

  return single_bond_radius[atomic_number1]
  + single_bond_radius[atomic_number2];
}

/* The following function frees an array packed molecule.  Note that it does */
/* Not free other, as that must be allocated as a 'special case' */
void free_molecule(atom *molecule) {
  atom *work_atom;

  if (molecule == NULL) {
  warn_out("NULL pointer passed to free_molecule(), this was probably "
       "unintended");
  return;
  }

  molecule -= get_atom_offset(molecule);

  for ( work_atom = molecule; work_atom != NULL; work_atom = work_atom->next )
{
  if (work_atom->label) { free(work_atom->label); }
  if (work_atom->coordinates) { free(work_atom->coordinates); }
  if (work_atom->bond) { free(work_atom->bond); }
  if (work_atom->bond_order) { free(work_atom->bond_order); }
  if (work_atom->qcode) { free(work_atom->qcode); }
  }

  /* And finally, free the array */
  free(molecule);

  return;
}

/* The following function prints a comma separated list of all the atoms */
/* in the molecule in the form "C,1.0,-1.0,0.0" or so.  It prints to the */
/* stream (which must be opened before the call) pointed to by the first */
/* argument */
void atom_print_xyz(FILE *out_stream, atom *molecule_member) {

  if (molecule_member == NULL) {
  error_exit("NULL passed to atom_print_xyz(), cannot continue");
  }

  while (molecule_member->previous != NULL) {
  molecule_member = molecule_member->previous;
  }

  /* And do the printing */
  while ( molecule_member != NULL ) {
  fprintf(out_stream, "%s,%f,%f,%f\n",
      molecule_member->label,
      molecule_member->coordinates[0],
      molecule_member->coordinates[1],
      molecule_member->coordinates[2] );

  molecule_member = molecule_member->next;
  }

  return;
}

/* The following function is very similar to atom_print_xyz, (in fact, */
/* it uses it) but it creates a blank .com file suitable for opening */
/* by gaussview */
void atom_print_com(FILE *out_stream, atom *molecule_member) {

  fprintf(out_stream, "No route\n\nNo Description\n\n0 1\n");
  atom_print_xyz(out_stream, molecule_member);

  return;
}

/* This function looks for bonds pointing to NULL that are under the */
/* valence of the atom passed to it. */
void atom_pack_this_atom_bonds(atom *some_atom) {

  int i, j;
  boolean pack_this, all_null;

  if (some_atom == NULL) {
  warn_out("NULL passed to atom_pack_this_atom_bonds()");
  }

  /* Some bonds within the valence may point to NULL, we need to shrink the */
  /* valences where needed */
  pack_this = no;
  for (i = 0; i < some_atom->valence; i++) {
  if (some_atom->bond[i] == NULL ) {
    pack_this = yes;
    all_null = true;
    for (j = i; j < (some_atom->valence - 1); j++) {
    if (some_atom->bond[j + 1] != NULL) {all_null = false;}
    /* Swap this bond and bond order with the next one for */
    /* All bonds (kind of a bubble swap) */
    some_atom->bond[j] = some_atom->bond[j + 1];
    some_atom->bond_order[j] =
      some_atom->bond_order[j + 1];
    }
    /* Check this bond again if all_null is false */
    if (all_null == false ) {
    some_atom->bond[some_atom->valence - 1] = NULL;
    i--;
    }
  }
  }
  if (pack_this == yes) { /* Repack the bonds and bond_orders */
  for (i = 0; i < some_atom->valence; i++) {
    if (some_atom->bond[i] == NULL) {
    some_atom->valence = i;
    }
  }
  if ( (some_atom->bond = realloc(some_atom->bond,
                some_atom->valence * sizeof(atom *) ) ) == NULL ) {
    warn_out("unable to repack bond vectors in "
          "atom_pack_this_atom_bonds(), may be fatal");
  }
  if ( (some_atom->bond_order =
      realloc(some_atom->bond_order, some_atom->valence *
          sizeof(float) ) ) == NULL ) {
```

```c
    warn_out("unable to repack bond orders in "
         "atom_pack_this_atom_bonds(), may be fatal");
  }
  }

  return;
}

/* This function simply calls atom_pack_this_atom() for each atom */
/* in the molecule passed to it */
void molecule_pack_all_bonds(atom *molecule) {

  int i, size;
  atom *new_molecule;

  new_molecule = molecule - get_atom_offset(molecule);
  size = molecule_get_size(new_molecule);

  for (i = 0; i < size; i++) {
  atom_pack_this_atom_bonds(new_molecule + i);
  }

  return;
}

/* This function is pretty simple, it gets the _size_ of the molecule. */
/* Note that this will be one greater than the offset of the last legal */
/* member, which means one must loop as: ( i = 0; i < size; i++ ). */
int molecule_get_size(atom *molecule) {

  int i = 0, j = 0;
  atom *work_atom;

  if (molecule == NULL) {
  warn_out("NULL pointer passed to atom_get_size");
  return -1;
  }

  /* Counting both directions takes less calculations than rewinding, and */
  /* counting from the beginning */
  for ( work_atom = molecule; work_atom != NULL;
    i++, work_atom = work_atom -> next ) {}

  for ( work_atom = molecule; work_atom != NULL;
    j++, work_atom = work_atom -> previous ) {}

  return i + j - 1;  /* Since we counted the atom at molecule twice */

}

/* The following function attempts to assign formal charges to the atom */
/* passed to it.  It only knows how to accurately do so for groups that */
/* been specified in the code.  It is designed to be very restrictive, */
/* so that it doesn't just bogusly give formal charges to unknown chemical */
/* groups.  It returns 1 if it changed the formal charge, and 0 if no */
/* change was made. */
int assign_formal_charge(atom *this_atom) {

  int i, j;
  float total_bond_order;
  boolean target_found;
  atom *work_atom[4];

  if (this_atom == NULL) {
  warn_out("NULL pointer passed to assign_formal_charge, may be fatal");
  }

  /* initializations */
  total_bond_order = get_total_bond_order(this_atom);
  work_atom[0] = NULL;
  work_atom[1] = NULL;
  work_atom[2] = NULL;
  work_atom[3] = NULL;

  /* While there is no code here yet, we need to check hypervalent atoms */
  /* first, since they are not well tested by is_atom_valence_full() */
  /* If the valence is already full, don't fiddle with the formal charges */

  if (is_atom_valence_full(this_atom)) { return 0; }

  /* Note that other groups can be added to the following section as needed. */
  /* Also, there is no way currently to distinguish between a carbocation */
  /* and a carbanion.  We attempt to do so by checking for three bonds and */
  /* seeing whether the charge is less than, equal to, or greater than 0.  */
  /* If the charge _is_ 0, it returns withough changing anything. */

  /* Handle carbon ions */
  if (this_atom->atomic_number == 6 && total_bond_order == 3.0 &&
    this_atom->valence == 3) {
  /* It must be some form of carbon ion */
  if (this_atom->charge == 0.0) { return 0; }
  if (this_atom->charge > 0.0) {
    this_atom->formal_charge = 1.0;
    return 1;
  } else {
    this_atom->formal_charge = -1.0;
    return 1;
  }
  }

  /* Handle alkoxides */
  if (this_atom->atomic_number == 8 && total_bond_order == 1.0 &&
    this_atom->valence == 1) {
  /* It must be an alkoxide */
  this_atom->formal_charge = -1.0;
  return 1;
  }

  /* Handle oxonium ions */
  if (this_atom->atomic_number == 8 && total_bond_order == 3.0 &&
    this_atom->valence == 3) {
  /* It is an oxonium */
  this_atom->formal_charge = 1.0;
  return 1;
  }

  /* Handle ammonium cations */
  if (this_atom->atomic_number == 7 && total_bond_order == 4.0 &&
    this_atom->valence == 4) {
  /* It is an ammonium */
  this_atom->formal_charge = 1.0;
  return 1;
  }

  /* Handle amide anions */
  if (this_atom->atomic_number == 7 && total_bond_order == 2.0 &&
    this_atom->valence == 2) {
  /* It is an amide */
  this_atom->formal_charge = -1.0;
  return 1;
  }

  /* Handle the nitro group or carboxylate possibility.  */
  if (this_atom->atomic_number == 8 && total_bond_order == 1.5 &&
    this_atom->valence == 1) {
  /* It must be a carboxylate or nitro, or some other resonance type */
  /* of oxygen, look at all of the atoms 2 bonds away for an oxygen, */
  /* and check/fix that valence as well.  In this case, target_found */
  /* that is being looked for is a nitrogen.  If found , it's formal */
  /* charge will also be fixed up */
  target_found = false;
  if ( this_atom->bond[0]->atomic_number == 7 &&
     get_total_bond_order(this_atom->bond[0]) == 4 &&
     this_atom->bond[0]->valence == 3 ) { target_found = true; }
  for (i = 0; i < this_atom->bond[0]->valence; i++) {
    if ( this_atom->bond[0]->bond[i]->atomic_number == 8 &&
       get_total_bond_order(this_atom->bond[0]->bond[i]) == 1.5 &&
       this_atom->bond[0]->bond[i]->valence == 1) {
    /* We found the partner! */
    this_atom->formal_charge = -0.5;
    this_atom->bond[0]->bond[i]->formal_charge = -0.5;
    /* And fix the nitrogen if it's appropriate */
    if (target_found) { this_atom->bond[0]->formal_charge = 1.0; }
    return 1;
    }
  }
  }
  /* Note:  The previous section handled a nitro group by the O side, we */
  /* also need to handle a nitro group by the N side */
  if (this_atom->atomic_number == 7 && total_bond_order == 4.0 &&
    this_atom->valence == 3) {
  /* It may just be a nitro group */
  j = 0;
  for (i = 0; i < this_atom->valence; i++) {
    if (this_atom->bond[i]->atomic_number == 8 &&
      get_total_bond_order(this_atom->bond[i]) == 1.5 &&
      this_atom->bond[i]->valence == 1 ) {
    /* It's a properly formatted O! */
    work_atom[j] = this_atom->bond[i];
    j++;
    }
  }
  /* Check to see if we found 2 O's that match */
  if ( work_atom[1] != NULL && work_atom[2] == NULL) {
    /* We found our group, do the appropriate changes and return 1 */
    this_atom->formal_charge = 1.0;
    work_atom[0]->formal_charge = -0.5;
    work_atom[1]->formal_charge = -0.5;
  } else {
    /* reset the work atom and continue */
    work_atom[0] = NULL;
  }
  }

  /* If none of the above requirements was satisfied, return the fact that */
  /* we have changed nothing */

  return 0;

}

/* The following function simply sums the bond orders of the atom passed */
/* to it, and returns that value */
float get_total_bond_order(atom *this_atom) {

  float total_bond_order = 0;
  int i;

  if (this_atom == NULL) {
  warn_out("NULL pointer passed to get_total_bond_order(), may be fatal");
  }

  for (i = 0; i < this_atom->valence; i++ ) {
  total_bond_order += this_atom->bond_order[i];
  }

  return total_bond_order;

  return 0;
}

/* The following function simply zeroes all of the states in the molecule */
void molecule_zero_states(atom *passed_atom) {

  for(passed_atom -= get_atom_offset(passed_atom); passed_atom;
    passed_atom = passed_atom->next) {
  passed_atom->state = 0;
  }

  return;
}

/* The following function simply maxes all of the states in the molecule */
void molecule_max_states(atom *passed_atom) {

  for(passed_atom -= get_atom_offset(passed_atom); passed_atom;
    passed_atom = passed_atom->next) {
  passed_atom->state = LONG_MAX;
```

```
  }

  return;
}


/* The following function simply returns a yes or no, as appropriate, if */
/* the atoms are connected */
boolean is_connected(atom* atom1, atom* atom2) {

  boolean connect1, connect2;
  int i;

  connect1 = connect2 = false;

  for (i = 0; i < atom1->valence; i++) {
  if ( atom1->bond[i] == atom2 ) { connect1 = true; break;}
  }
  for (i = 0; i < atom2->valence; i++) {
  if ( atom2->bond[i] == atom1 ) { connect2 = true; break;}
  }

  if ( connect1 && connect2 ) { return yes; }

  /* The following line acts as an xor */
  if ( connect1 || connect2 ) {
  warn_out("Non-mutually connected (i.e., one is connected to the "
       "other, but the other isn't connected to one ... (is that "
       "clear enough?) atoms passed to is_connected");
  }

  return no;
}

/* This is just an alias for is_connected */
boolean is_bonded(atom* atom1, atom* atom2) {

  return(is_connected(atom1, atom2));

}

/* The following routine simply does a few checks to see if we have a */
/* carbon for which we need to assign a stereochemical descriptor */
boolean is_asymmetric_carbon(atom *some_atom) {

  int i, j, k;
  float slop = 0.000000000001;
  /* It may be much better to define this slop in a #define statement, as */
  /* it's also present in qdb_shared_functions.c, and may show up in other */
  /* float comparison functions */

  if (some_atom == NULL) {
  warn_out("Null pointer passed to is_asymmetric_carbon(), this was "
       "almost certainly uninteded, but is not immediately fatal");
  return no;
  }

  if (some_atom->atomic_number != 6) {
  return no;
  }

  /* Finally, make sure there's four groups */
  if (some_atom->valence != 4) {
  return no;
  }

  /* Ok, now we have a carbon, and should be relatively error free, on to */
  /* the real work */
  for(i = 0; i < some_atom->valence; i++) {
  for(j = 0; j < i; j++) {
    for(k = 0; k < QDEPTH; k++) {
    if(fabs( some_atom->bond[i]->qcode[k]
        - some_atom->bond[j]->qcode[k]) > slop) {
      k = QDEPTH + 1;
    }
    if(k <= QDEPTH) {
      return no;
    }
    }
  }
  }

  return yes;
}

/* The following function is a wrapper for is_asymmetric_carbon(), but */
/* it returns an int, which is what recurse_molecule_do() requires */
int i_is_asymmetric_carbon(atom *some_atom) {

  return is_asymmetric_carbon(some_atom) ? 1 : 0;

}

/* The following function assigns the 'qcode stereochemical descriptor' */
/* for the atom it receives.  It can safely be called on any atom, since */
/* it calls is_asymmetric_carbon() before it does it's work.  Note that */
/* it will _always_ overwrite whatever is within the atom passed to */
/* it, even if this means overwriting it with a '\0'.  Relatively thorough */
/* auditing of this function implies that it has no memory leaks (the */
/* author uses the dmalloc library for this purpose */
void assign_q_s_descriptor(atom *some_atom) {

  atom *atom_p;
  atom *sorted[4];   /* We assume we'll only have to sort 4 groups */
  vector_d atomcc, atom1c, atom2c, atom3c, atom4c, work_vector;
  matrix_d transform;
  boolean all_sorted = no, made_correction = no;
  int i;

  /* is_asymmetric_carbon traps NULL pointers, so we'll let that function */
  /* handle that case */
  if ( is_asymmetric_carbon( some_atom ) == no ) {
  some_atom->s_descriptor = '\0';
  return;
  }

  /* The first step is to prioritize the four groups.  This will be done */
  /* with yet another function, so it will be easy (as easy as the task */
  /* can be) to convert to another prioritization scheme, i.e. CIP */
  /* prioritization */

  /* Because I'm not interested in implementing a complex sort algoritm, */
  /* and we're sorting very few things, I'll use a simple one directonal */
  /* bubble sort */

  for ( i = 0; i < some_atom->valence; i++ ) {
  sorted[i] = some_atom->bond[i];
  }

  i = 0;
  while (! all_sorted) {
  if ( gimme_higher_priority ( sorted[i], sorted[i + 1] ) == sorted[i] ) {
    /* Then the lower one has higher priority, and should be moved */
    /* up (swapped) with the upper value */
    atom_p = sorted[i];
    sorted[i] = sorted[i + 1];
    sorted[i + 1] = atom_p;

    /* And we're not sorted since we had to do a switch */
    made_correction = yes;
  }

  if ( i == some_atom->valence - 2 ) {
    i = 0;

    /* We may be all sorted */
    if ( made_correction == no ) {
    all_sorted = yes;
    } else {
    /* Since we're at the end of a 'pass', reset made_correction */
    made_correction = no;
    }
  } else {
    i++;
  }
  }

  /* Ok, it looks like the list (sorted[]) was sorted properly.  Now we */
  /* need to transform the coordinates of the four groups to an orientation */
  /* where the lowest priority is facing back, the central atom is at */
  /* 0, 0, 0, and the second lowest priority is on the x axis.  for this */
  /* task, we'll be taking advantage of the vector routines, and the */
  /* xformdipole program I wrote earlier in my work */

  /* The first step is to initialize all of the members of the 5 atom */
  /* variables declared at the beginning of the function.  We will be */
  /* transforming _copies_ of the atoms coordinates */

  if (!( atomcc = (vector_d)malloc(3 * VECDVALSIZE) ) ) {
  error_exit(
       "Cannot initialize space for new vector in assign_q_s_descriptor()"
       );
  }

  if (!( atom1c = (vector_d)malloc(3 * VECDVALSIZE) ) ) {
  error_exit(
       "Cannot initialize space for new vector in assign_q_s_descriptor()"
       );
  }

  if (!( atom2c = (vector_d)malloc(3 * VECDVALSIZE) ) ) {
  error_exit(
       "Cannot initialize space for new vector in assign_q_s_descriptor()"
       );
  }

  if (!( atom3c = (vector_d)malloc(3 * VECDVALSIZE) ) ) {
  error_exit(
       "Cannot initialize space for new vector in assign_q_s_descriptor()"
       );
  }

  if (!( atom4c = (vector_d)malloc(3 * VECDVALSIZE) ) ) {
  error_exit(
       "Cannot initialize space for new vector in assign_q_s_descriptor()"
       );
  }

  /* Initialize the transform matrix space */
  if (!( transform = (matrix_d)malloc(3 * sizeof(vector_d) ) ) ) {
  error_exit(
       "Cannot initialize space for new matrix (of 3 vectors) in "
       "assign_q_s_descriptor()" );
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  /* Copy the relevant values */
  for ( i = 0; i < 3; i++ ) {
  atomcc[i] = some_atom->coordinates[i];
  }

  for ( i = 0; i < 3; i++ ) {
  atom1c[i] = sorted[0]->coordinates[i];
  }

  for ( i = 0; i < 3; i++ ) {
  atom2c[i] = sorted[1]->coordinates[i];
  }

  for ( i = 0; i < 3; i++ ) {
  atom3c[i] = sorted[2]->coordinates[i];
  }

  for ( i = 0; i < 3; i++ ) {
  atom4c[i] = sorted[3]->coordinates[i];
```

```
  }

  /* All of the values are in place, and this is where we start stealing */
  /* code (significantly) from xformdipole.c */

  /* vec_subtract allocates new memory */

  transform[0] = vec_subtract(atomcc, atom1c, 3);
  transform[1] = vec_subtract(atomcc, atom2c, 3);

  /* Now orthogonalize the second vector */
  work_vector = transform[1];
  transform[1] = vec_orthogonalize(transform[0], transform[1], 3);
  free(work_vector);

  /* Normalize both */
  for( i = 0; i < 2; i ++) {
  work_vector = vec_normalize(transform[i], 3);
  free(transform[i]);
  transform[i] = work_vector;
  }

  /* And get the cross product for the last row of the tranform matrix*/
  transform[2] = vec3_cross_product(transform[0], transform[1]);

  /* Translate all of the vectors to be centered on atomcc */
  for (i = 0; i < 3; i++) {
  atom1c[i] -= atomcc[i];
  atom2c[i] -= atomcc[i];
  atom3c[i] -= atomcc[i];
  atom4c[i] -= atomcc[i];
  atomcc[i] = 0.0;
  }

  /* For debugging purposess */
  /* printf("Before:\n"); */
  /* printf("Center atom: %f\t%f\t%f\n", atomcc[0], atomcc[1], atomcc[2]); */
  /* printf("Atom 1:      %f\t%f\t%f\n", atom1c[0], atom1c[1], atom1c[2]); */
  /* printf("Atom 2:      %f\t%f\t%f\n", atom2c[0], atom2c[1], atom2c[2]); */
  /* printf("Atom 3:      %f\t%f\t%f\n", atom3c[0], atom3c[1], atom3c[2]); */
  /* printf("Atom 4:      %f\t%f\t%f\n", atom4c[0], atom4c[1], atom4c[2]); */
  /* printf("\n"); */

  /* Finally, do all 5 transforms */
  work_vector = vec_transform(transform, atomcc, 3);
  free(atomcc);
  atomcc = work_vector;

  work_vector = vec_transform(transform, atom1c, 3);
  free(atom1c);
  atom1c = work_vector;

  work_vector = vec_transform(transform, atom2c, 3);
  free(atom2c);
  atom2c = work_vector;

  work_vector = vec_transform(transform, atom3c, 3);
  free(atom3c);
  atom3c = work_vector;

  work_vector = vec_transform(transform, atom4c, 3);
  free(atom4c);
  atom4c = work_vector;

  /* For debugging purposess */
  /* printf("After:\n"); */
  /* printf("Center atom: %f\t%f\t%f\n", atomcc[0], atomcc[1], atomcc[2]); */
  /* printf("Atom 1:      %f\t%f\t%f\n", atom1c[0], atom1c[1], atom1c[2]); */
  /* printf("Atom 2:      %f\t%f\t%f\n", atom2c[0], atom2c[1], atom2c[2]); */
  /* printf("Atom 3:      %f\t%f\t%f\n", atom3c[0], atom3c[1], atom3c[2]); */
  /* printf("Atom 4:      %f\t%f\t%f\n", atom4c[0], atom4c[1], atom4c[2]); */
  /* printf("\n"); */

  /* The following is the big workhorse.  Uncomment the previous sections */
  /* to see information on the tranformed coordinates */
  if (atom3c[2] > 0 && atom4c[2] < 0) {
  some_atom->s_descriptor = 's';
  } else if (atom3c[2] < 0 && atom4c[2] > 0) {
  some_atom->s_descriptor = 'r';
  } else {
  error_exit("Unphysical geometry about central carbon passed to "
        "assign_q_s_descriptor()");
  }

  /* Before we leave, free all newly allocated memory */
  free(atomcc);
  free(atom1c);
  free(atom2c);
  free(atom3c);
  free(atom4c);
  for ( i = 0; i < 3; i++) {
  free(transform[i]);
  }
  free(transform);

  return;
}

atom *gimme_higher_priority(atom *atom1, atom *atom2) {

  float slop = 0.000000000001;
  int i;

  for(i = 0; i < QDEPTH; i++) {
  if(fabs(atom1->qcode[i] - atom2->qcode[i]) > slop) {
    /* We're ready to return! */
    if (atom1->qcode[i] > atom2->qcode[i]) {
    return atom1;
    } else {
    return atom2;
    }
  }
  }

  /* If we get here, there's an error, return NULL */
  return NULL;
}

/* The following function initializes space for an atom and all of its */
/* allocated portions of the atom.  It initializes them to 0,'\0' or */
/* NULL, as appropriate -- and assumes the label will be no longer than */
/* 3 characters */
atom *initialize_blank_atom(void) {

  atom *some_atom;

  if ( ( some_atom = malloc(sizeof(atom) ) ) == NULL ) {
  error_exit(
        "Cannot allocate memory for new atom in initialize_blank_atom()");
  }

  if ( (some_atom->label = malloc(4 * sizeof(char) ) ) == NULL ) {
  error_exit(
        "Cannot allocate memory for atom label in initialize_blank_atom()");
  }

  if(!( some_atom->coordinates = malloc(3 * VECDVALSIZE) ) ) {
  error_exit(
        "Cannot allocate memory for coordinates in "
        "initialize_blank_atom()");
  }

  if( ( some_atom->bond_order = malloc(sizeof(float) ) ) == NULL) {
    error_exit(
        "Cannot allocate memory for inital bond_order  "
        "initialize_blank_atom()");
  }

  some_atom->label[0] = '\0';
  some_atom->atomic_number = 0;
  some_atom->coordinates[0] = 0.0;
  some_atom->coordinates[1] = 0.0;
  some_atom->coordinates[2] = 0.0;
  some_atom->valence = 0;
  some_atom->bond = NULL;
  some_atom->bond_order[0] = 0;
  some_atom->formal_charge = 0.0;
  some_atom->qcode = NULL;
  some_atom->charge = 0.0;
  some_atom->s_descriptor = '\0';
  some_atom->other = NULL;
  some_atom->next = NULL;
  some_atom->previous = NULL;

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  return some_atom;
}

/* The following function does whatever is asked for in the first argument */
/* (a function pointer that must return int, and take one atom * as an */
/* argument) to the molecule, beginning at the atom in the second */
/* argument, to all of the atoms within a range of depth (it will always */
/* do something to the first atom, regardless of whether or not depth is */
/* >= 0.  Note that this function is a wrapper for the workhorse, only */
/* because we need to initialize the states of the molecules.  Note that */
/* The following two functions use a (recently) file scope variable */
/* since the workhorse needs to know about the maximum depth, as we want */
/* to start marking the beginning atom from 0, not from the original depth */
/* Note that after testing and revision, do_function is done exactly */
/* _once_ to every atom within range specified by depth.  It is not */
/* guaranteed that it will be done in any order, only that it will be */
/* done exactly once.  */
/* After some usage of the function, it has become apparent that there */
/* may be several 'ways' a user would like the function to behave, i.e., */
/* sometimes, the user wants to use the states afterwords, sometimes they */
/* don't.  Other times, they may not need the side effect of adding the */
/* members to the list.  The mode can be formed manually, or by bitwise */
/* or'ing the following options together: */
/* RMD_SAVESTATES(1)  <--- save and restore states */
/* RMD_NOTOUCHLIST(2) <--- do not touch the list provided by */
/*                         atom_list_manage() */
/* RMD_CLEANMEM(4)    <--- release all memory before returning, i.e., */
/*                         the states and atom_list_manage lists */
/* RMD_NOINITSTATES(8)<---  Don't initialize the states yourself, */
/*                          assume the caller has already done so */
/* Also, a call to this function has the side effect */
/* of marking the atoms' states as follows: */
/* Untouched atoms states are LONG_MAX */
/* other atoms are marked with 'how far' they are from the original atom. */
/* These, of course, are not noticeable by the calling environment if */
/* the function is called with RMD_SAVESTATES.  Note that not all of the */
/* mode flags have been thoroughly tested */
static int max_i_recurse_depth;

int recurse_molecule_do( int (*do_function)(atom *), atom *this_atom,
            int depth, int mode) {

  static long int *old_states;
  int retval;

  if ( mode & RMD_SAVESTATES ) {
  old_states = save_states(this_atom);
  }

  if ( !(mode & RMD_NOINITSTATES) ) {
  /* Set the molecule states to prepare for recursion */
  molecule_max_states(this_atom);
  }

  /* Empty the atom_list (accessed via atom_list_manage), as we'll keep */
  /* a list of all atoms who returned non-zero in do_function */
  if ( !(mode & RMD_NOTOUCHLIST ) ) {
  atom_list_manage(this_atom, A_CLEAR);
  }
```

```
  max_i_recurse_depth = depth;

  retval =  recurse_molecule_do_core(do_function, this_atom, depth, mode);

  if ( mode & RMD_SAVESTATES ) {
  restore_states(this_atom, old_states);
  old_states = NULL;
  }

  if ( mode & RMD_CLEANMEM ) {
  if (old_states) {
    free(old_states);
  }
  atom_list_manage(this_atom, A_CLEAR);
  }

  return retval;
}

/* The following function actually does the recursive work portion of the */
/* algorithm outlined for recurse_molecule_do() */
int recurse_molecule_do_core( int (*do_function)(atom *), atom *this_atom,
                int depth, int mode) {

  int return_value = 0;
  /* I'm not sure if declaring these static results in any speed increase, */
  /* but the idea is that the function doesn't really need to allocate */
  /* a new one every time it visits, as re-using the old one should be */
  /* just fine (since we always initialize it */
  int i;
  static int this_state;

  /* Initialization */
  this_state = max_i_recurse_depth - depth;

  /* End the recursion if we're deeper than another visit here */
  if ( this_state >= this_atom->state ) { return 0; }

  /* Do the deed expressed in do_function, but only if it hasn't been */
  /* done already, i.e., the state is still LONG_MAX */
  if (this_atom->state == LONG_MAX) {
  i = do_function(this_atom);
  if ( i  && !(mode & RMD_NOTOUCHLIST) ) {
    atom_list_manage(this_atom, A_PUSH);
  }
  return_value += i;
  }

  /* Prepare for the next iteration */
  this_atom->state = this_state;
  depth--;

  /* Since we wanted to do that 'special' something, even if the depth passed
*/
  /* to us was zero, we waited until now for the depth based  bail-out */
  /* condition */
  if ( depth < 0 ) { return return_value; }

  /* Now, we do the main bond following portion */
  for ( i = 0; i < this_atom->valence; i++ ) {
  /* Remember, we already decremented depth */
  /* Only visit the next atom if we have to */
  if (this_atom->bond[i]->state > this_atom->state ||
    this_atom->bond[i]->state == LONG_MAX ) {
    return_value += recurse_molecule_do_core(do_function,
                      this_atom->bond[i], depth, mode);
  }
  }

  return return_value;
}

/* The following function simply returns true or false (which are */
/* conveniently 0 and 1 for the sake of passing through */
/* recurse_molecule_do) based on whether the s_descriptor field of */
/* the atom is non '\0' or not */
boolean does_it_have_s_descriptor(atom *some_atom) {

  if ( some_atom->s_descriptor != '\0' ) {
  return true;
  } else {
  return false;
  }

  /* And, to keep some compilers happy */
  return false;

}

/* The following is a wrapper for the function with the similar name, */
/* except it returns an int */
int i_does_it_have_s_descriptor(atom *some_atom) {

  return does_it_have_s_descriptor(some_atom) ? 1 : 0;

}


/* The following function maintains a static internal list, that can be */
/* manipulated with various op_codes, but cannot be directly accessed. */
/* Various functions may use it to keep a list of atoms handy.  Note that */
/* it will leave memory allocated until requested to clean itself, but */
/* it should be leak free itself.  It exits catastrophically on any  */
/* memory errors. */
/* The op codes are as follows: (the defined constants are written below */
/* the numbers */
/* 0   ->   Free any memory in the function if there is any (Note: all */
/* A_CLEAR  stored information is lost! */
/* 1   ->   An odd way to pass an integer when the return type is */
/* A_COUNT  (atom *), it returns the atom whose offset (via */
/*          get_atom_offset() ) is the same as the number of atoms in the */
/*          list.  If the supplied molecule is not big enough to report */
/*          this value, it returns NULL.  If the list is empty, it returns */
/*          the molecule's base pointer */
/* 2   ->   Emulates perl's push function, returns push'd atom on success */
/* A_PUSH     */
/* 3   ->   Emulates perl's pop function, returns pop'ed atom *, or NULL */
/* A_POP         if the list is empty */
/* 4   ->   Emulates perl's shift function, returns shift'ed atom on success */
/* A_SHIFT    */
/* 5   ->   Emulates perl's unshift function, returns unshift'ed atom *, */
/* A_UNSHIFT   or NULL if the list is empty */
/* Other op codes can be added easily, I have no intention of emulating */
/* perl's splice function for now.  Note that push'ing and pop'ing will */
/* always be more efficient than shift'ing and unshift'int */

atom *atom_list_manage(atom *some_atom, int op_code) {

  int i;
  atom *work_atom, *molecule_base;
  static int last_atom_index = -1;
  static atom **atom_list = NULL;

  if (op_code == 1 && some_atom == NULL) {
  if (some_atom == NULL) {
    /* The caller messed up, exit immediately */
    error_exit("NULL pointer received by atom_list_manage() on "
         "a count call, this makes no sense");
  }
  }

  switch (op_code) {
  case 0:
  /* Empty list */
  if (atom_list != NULL) {
    free(atom_list);
    atom_list = NULL;
    last_atom_index = -1;
  }
  if (last_atom_index != -1) {
    error_exit("last_atom_index is hosed (not -1) in "
         "atom_list_manage()");
  }
  return NULL;
  case 1:
  /* Return # of elements in list */
  if (last_atom_index == -1) {
    return some_atom - get_atom_offset(some_atom);
  }

  molecule_base = work_atom = some_atom - get_atom_offset(some_atom);
  for(i = 0; work_atom[i].next; i++) { }
  /* i is now the index to the last atom in the molecule */
  if ( last_atom_index < i ) {
    return molecule_base + last_atom_index + 1;
  } else {
    return NULL;
  }
  case 2:
  /* Push */

  if (some_atom == NULL) { return NULL; }

  last_atom_index++;
  if ( ( atom_list = realloc(atom_list, (last_atom_index + 1) *
               sizeof(atom*) ) ) == NULL ) {
    error_exit("Unable to expand atom_list in atom_list_manage()"
         "(push)");
  }

  atom_list[last_atom_index] = some_atom;
  return some_atom;
  case 3:
  /* Pop */
  if (last_atom_index == -1) {
    return NULL;
  }
  work_atom = atom_list[last_atom_index];
  last_atom_index--;
  if (last_atom_index == -1) {
    free(atom_list);
    atom_list = NULL;
  } else if ( ( atom_list = realloc(atom_list, (last_atom_index + 1) *
                  sizeof(atom*) ) ) == NULL ) {
    error_exit("Unable to shrink atom_list in atom_list_manage()"
         "(pop)");
  }
  return work_atom;
  case 4:
  /* Shift */

  if (some_atom == NULL) { return NULL; }

  last_atom_index++;
  if ( ( atom_list = realloc(atom_list, (last_atom_index + 1) *
               sizeof(atom*) ) ) == NULL ) {
    error_exit("Unable to expand atom_list in atom_list_manage()"
         "(shift)");
  }

  for ( i = last_atom_index; i > 0;  i-- ) {
    atom_list[i] = atom_list[i - 1];
  }
  atom_list[0] = some_atom;
  return some_atom;
  case 5:
  /* Unshift */
  if (last_atom_index == -1) {
    return NULL;
  }
  work_atom = atom_list[0];
  for ( i = 0; i < last_atom_index; i++ ) {
    atom_list[i] = atom_list[i + 1];
  }
  last_atom_index--;
  if (last_atom_index == -1) {
    free(atom_list);
    atom_list = NULL;
```

```
  } else if ( ( atom_list = realloc(atom_list, (last_atom_index + 1) *
                  sizeof(atom*) ) ) == NULL ) {
    error_exit("Unable to shrink atom_list in atom_list_manage()"
         "(unshift)");
  }
  return work_atom;

  default:
  /* Error case */
  error_exit("Unknown op_code passed to atom_list_manage()");
  }

  /* And ... to keep some compilers happy */
  return NULL;
}


/* The following function tries to assign bond orders where they're not */
/* provided for, and also assigns formal charges.  Finally, it verifies */
/* all of the valences.  It will make one pass through, even if it fails */
/* early */
boolean verify_molecule_connectivity(atom *some_atom) {

  atom *molecule_base, *work_atom;
  boolean return_value;
  int i, j, molecule_size = 0;
  float this_bond_order;
  char work_string[MAXSTR];

  if ( ! some_atom ) {
  error_exit("NULL atom passed to verify_molecule_connectivity()");
  }

  molecule_base = some_atom - get_atom_offset(some_atom);

  for (work_atom = molecule_base; work_atom; work_atom = work_atom->next) {
  molecule_size++;
  }

  for (i = 0; i < molecule_size ; i++ ) {
  /* Only search for a partner if this atom doesn't have a full */
  /* valence already */
  if (!is_atom_valence_full(molecule_base + i)) {
    for ( j = 0; j < i; j++ ) {
    this_bond_order =
      get_atom_to_atom_bond_order(molecule_base + i,
                  molecule_base +j );
    if(this_bond_order != -1.0 ) {
      atom_connect(molecule_base + i, molecule_base + j,
           this_bond_order);
    }
    }
  }
  }

  /* Now that we have all of the bonds, fill the valences */

  /* We need to assign formal charges now.  This routine is a compromise, */
  /* in that it specifically recognizes groups that have been programmed */
  /* in, but cannot predict strange situations very well.  */

  /* Verify that valences are correctly filled */
  for (work_atom = molecule_base; work_atom; work_atom = work_atom->next) {
  if (!is_atom_valence_full(work_atom)) {
    /* Try to assign formal charges before announcing an error */
    assign_formal_charge(work_atom);

    /* Also, try to repair the bond orders/connectivity */
    repair_connectivity(work_atom);

    /* If the atom is still not valence saturated, announce a warning */
    if (!is_atom_valence_full(work_atom)) {
    i = get_atom_offset(work_atom);
    sprintf(work_string, "atom at offset %d (gaussview %d) does "
        "not have a properly filled\nvalence.", i, i+1);
    warn_out(work_string);
    }
  }
  }

  /* I'm not certain why I haven't used this in the function, it can */
  /* be investigated at a later time */
  return_value = true;

  return return_value;

}

/* The following function does it's best to 'repair' connectivity */
/* orginally assigned by functions such as get_bond_order, etc.  */
/* Unfortunately, using bonding distances has many odd special */
/* cases, and this function tries all the trick's it's taught to */
/* 'fix' the bond order it's given.  It takes a single atom as an */
/* argument, and returns nothing, since it also guarantees nothing */
/* about its decisions.  Generally speaking, it should be used with */
/* the utmost care.  It should also _not_ step on the toes of  */
/* assign_formal_charges() */
void repair_connectivity(atom *some_atom) {

  int i, j, bond_marker;
  atom *work_atom;
  float distance, max_distance;

  /* This function is simply a set of rules, the switch is perfect for this */
  /* task */

  /* After tweaking with get_bond_order() (lengthening C-H bonds, and */
  /* letting oxygen be even more sloppy in it's bonding --- we'll have to */
  /* watch the tolerance for that), the only problems we seem to have is */
  /* for carbons that are in single bond positions of conjugated systems */
  /* This is an easy case to deal with very specifically. */

  switch (some_atom->atomic_number) {
```

```
  case 6:
  /* We are very stringent about what we'll actually 'fix' here */
  max_distance = 0.0;
  for (i = 0; i < some_atom->valence; i++) {
    if (some_atom->bond[i]->atomic_number == 6 &&
      get_total_bond_order(some_atom->bond[i]) > 4 &&
      some_atom->bond_order[i] == 1.5 ) {

    /* We've found the hypervalent carbon partner to us (maybe), */
    /* find the farthest carbon from us, and make sure that's */
    /* the same one we found.  Note that we couldn't check to */
    /* see that the remote atom's bond order back to us is 1.5, */
    /* this capability may need to be added later, though I don't */
    /* think so */

    for(j = 0; j < some_atom->valence; j++) {
      if (some_atom->bond[i]->atomic_number == 6) {
      distance = sqrt(
            (some_atom->coordinates[0] -
             some_atom->bond[j]->coordinates[0]) *
            (some_atom->coordinates[0] -
             some_atom->bond[j]->coordinates[0]) +
            (some_atom->coordinates[1] -
             some_atom->bond[j]->coordinates[1]) *
            (some_atom->coordinates[1] -
             some_atom->bond[j]->coordinates[1]) +
            (some_atom->coordinates[2] -
             some_atom->bond[j]->coordinates[2]) *
            (some_atom->coordinates[2] -
             some_atom->bond[j]->coordinates[2])
            );
      if (distance > max_distance) {
        max_distance = distance;
        work_atom = some_atom->bond[j];
      }
      }

      /* Is it the same one we found before? */
      if (some_atom->bond[i] == work_atom) {
      /* We have permission to fix these two */

      /* Find work_atom's pointer back here */
      for (j = 0; j < work_atom->valence; j++) {
        if (work_atom->bond[j] == some_atom) {
        bond_marker = j;
        j = work_atom->valence;
        break;
        }
      }
      /* And ... fix them */
      some_atom->bond_order[i] = 1;
      work_atom->bond_order[bond_marker] = 1;
      }
    }
    } else if (some_atom->bond[i]->atomic_number == 7 &&
        get_total_bond_order(some_atom->bond[i]) == 3.5 &&
        some_atom->bond_order[i] == 1.5 ) {
    /* Sometimes, if a carbon is in a conjugated system, and */
    /* is connected to a nitrogen, the bond will be anomalously */
    /* short.  In this case, we can do the repair.  We will */
    /* Not handle the case for nitrogen, since this side will */
    /* solve it.  */

    work_atom = some_atom->bond[i];

    /* Find work_atom's pointer back here */
    for (j = 0; j < work_atom->valence; j++) {
      if (work_atom->bond[j] == some_atom) {
      bond_marker = j;
      j = work_atom->valence;
      break;
      }
    }
    /* And ... fix both of them */
    some_atom->bond_order[i] = 1;
    work_atom->bond_order[bond_marker] = 1;
    }
  }
  break;

  default:
  /* Do nothing */
  ; /* Strangely, the compilers on the DEC cluster complained about not */
    /* having a statement between the default: and the closing brace, */
    /* so there it is */
  }

  return;
}

/* This function has a very simple and obvious purpose :-) */
boolean is_methyl_group(atom *some_atom) {

  int h_count = 0, i;

  if (some_atom->atomic_number != 6) { return false; }
  for ( i = 0; i < some_atom->valence; i++ ) {
  if(some_atom->bond[i]->atomic_number == 1) { h_count++; }
  }

  if ( h_count != 3 ) { return false; }

  return true;
}

/* The following function uses relatively simple criteria, and can be */
/* expanded to be more 'correct' in the future, if needed.  */
boolean is_aromatic(atom *some_atom) {

  int i, j;

  j = 0;
  for (i = 0; i < some_atom->valence; i++) {
  if ( some_atom->bond_order[i] == 1.5 ) { j++; }
```

```c
  }

  if ( j == 2 ) { return true; }

  return false;

}

/* The following function saves the states of all of the atoms in the */
/* molecule, to be restored later by it's sister function, restore_states */
/* It returns the requested pointer on success, or NULL if it fails, */
/* (can't allocate the requested memory) */
/* Warning:  This function allocates new memory, which must eventually be */
/* freed */
long int *save_states(atom *some_atom) {

  /* Variable declaration */
  long int *return_array;
  int molecule_size, i;
  atom *work_atom, *molecule_base;

  /* Error checking */
  if ( !some_atom ) {
  warn_out("NULL pointer passed to save_states(), this may be fatal");
  return NULL;
  }

  /* Initialization */
  molecule_base = some_atom - get_atom_offset(some_atom);

  molecule_size = 0;
  for (work_atom = molecule_base; work_atom; work_atom = work_atom->next) {
  molecule_size++;
  }

  /* Molecule size is now the number of atoms in the molecule */
  if ( (return_array = malloc( (molecule_size + 1) * sizeof (long int) ) )
     == NULL ) {
  warn_out("Unable to allocate memory for array of long ints in "
       "save_states()");
  }


  /* Mark the first element of the array with the number of atoms */
  return_array[0] = molecule_size;

  /* Finally, copy over all the states */
  for ( i = 0; i < molecule_size; i++ ) {
  /* Remember, return_array is 1 based, since element 0 holds its size */
  return_array[ i + 1 ] = molecule_base[i].state;
  }

  /* And, return it */
  return return_array;
}

/* The following function restores the given states to a molecule.  Note */
/* that if a set of states is given it that is different in size than */
/* the molecule passed to it, it will return 0, otherwise, on success, */
/* it returns 1.  It also frees the memory when it's done.  */
int restore_states(atom *some_atom, long int *old_states) {

  /* Variable declaration */
  int molecule_size, i;
  atom *molecule_base;

  /* Error checking */
  if ( !some_atom ) {
  warn_out("NULL atom passed to restore_states(), this may be fatal, "
       "returning 0");
  return 0;
  }

  if ( !old_states ) {
  warn_out("NULL list passed to restore_states(), this may be fatal, "
       "returning 0");
  return 0;
  }

  /* Initialization */
  molecule_base = some_atom - get_atom_offset(some_atom);
  molecule_size = old_states[0];

  /* Finally, copy over all the states */
  for ( i = 0; i < molecule_size; i++ ) {
  /* Remember, return_array is 1 based, since element 0 holds its size */
  molecule_base[i].state = old_states[ i + 1 ];
  }

  /* And, before we return, free old_states */
  free(old_states);

  return 1;
};

/* The following function gives a very powerful way to edit a molecule. */
/* What it does it take the molecule as given by to_keep, and begins */
/* by moving down the bond given by bond_number, deleting any atoms it */
/* encounters along that path.  If the atom tokeep is a member of a  */
/* ring (of any size), nothing will be deleted.  Note that the  */
/* function doesn't _actually_ delete the atoms, as the index of the */
/* atoms may be important to the calling function.  What it does is set */
/* the atomic number of all of the atoms to be deleted to UINT_MAX. */
/* It does not set the bonds to the group to be deleted to NULL, and does */
/* _not_ repack the bonds.  To 'finish' the job, the caller should (after */
/* they don't need the old indexes anymore) call repack_molecule(). */
/* Note that it does not initialize the states, as the calling environment */
/* is responsible for that task.  Unfortunately, in order to mark */
/* multiple parts of the molecule for deletion, this was a necessary */
/* step, since the states need to be left between subsequent calls. */
void delete_molecule_group( atom *tokeep, int bond_number ) {

  /* Declarations */
  int molecule_size;
  atom *tokeepbase, *work_atom;

  /* Error checking */
  if (tokeep == NULL) {
  warn_out("NULL atom pointer passed to delete_molecule_group(), "
       "returning after doing nothing");
  return;
  }

  if (bond_number < 0 || bond_number >= tokeep->valence) {
  warn_out("Invalid bond_number passed to delete_molecule_group(), "
       "returning after doing nothing");
  }

  /* Before we go on with the work of marking everything, see if the */
  /* fragment in question has already been deleted */
  if (tokeep->bond[bond_number]->state != LONG_MAX ) {
  return;
  }

  /* Initializations */
  molecule_size = molecule_get_size(tokeep);
  tokeepbase = tokeep - get_atom_offset(tokeep);

  /* Now, we mark the the atom we want to keep as already visited */
  tokeep->state = -1;

  /* Now, we can call recurse_molecule_do, and let it do the work */
  recurse_molecule_do ( &i_noop_a, tokeep->bond[bond_number],
           molecule_size, RMD_NOINITSTATES);

  for (work_atom = tokeepbase; work_atom; work_atom = work_atom->next ) {
  if (work_atom->state != LONG_MAX && work_atom->state != -1 ) {
    work_atom->atomic_number = UINT_MAX;

    /* And mark all of these atoms with -1 so subsequent calls don't */
    /* visit them */
    work_atom->state = -1;
  }
  }

  /* And, since we're keeping tokeep, do so */
  tokeep->state = LONG_MAX;

  return;
}

/* The following function does absolutely nothing, and always returns 0.  */
/* it is useful for using recurse_molecule_do for nothing but the side */
/* effects of setting the states. */
int i_noop_a (atom *nothing) {
  return 0;
}

/* The following function takes the molecule given by old_molecule, and */
/* repacks it into a new array, which it returns.  It has several tasks. */
/* First, it copies _only_ the atoms who do not have thier atomic numbers */
/* set to UINT_MAX into a new molecule.  It then has the task of re_aligning */
/* all of the bond pointers, as well as the next and last pointers.  For */
/* each of the atoms that are deleted, it needs to free the label, bond*, */
/* qcode, and coordinates.  It then frees the old array of atoms.  Finally, */
/* it repacks all of the bonds.  Note that some atoms may be left with */
/* valences that are no longer full.  In these cases, the 'messy' bonds */
/* will need to be cleaned up elsewhere */
atom *repack_molecule(atom *old_molecule) {

  /* Declarations */
  atom *work_atom, *old_molecule_base, *new_molecule;
  int new_size = 0, new_molecule_index = 0;

  /* Error checking */
  if (old_molecule == NULL) {
  warn_out("NULL pointer passed to repack_molecule(), doing nothing");
  return NULL;
  }

  /* Initializations */
  old_molecule_base = old_molecule - get_atom_offset(old_molecule);

  for (work_atom = old_molecule_base; work_atom; work_atom = work_atom->next) {
  if ( work_atom->atomic_number != UINT_MAX) {
    new_size++;
  }
  }

  /* Get a new molecule */
  if ( ( new_molecule = malloc( new_size * sizeof(atom) ) ) == NULL ) {
  warn_out("Unable to allocate space for the new molecule in "
       "repack_molecule(), returning original unchanged");
  return old_molecule;
  }

  /* Now, go through the old molecule and do all of the work appropriate */
  /* to either keeping or discarding the original molecule */
  for (work_atom = old_molecule_base; work_atom; work_atom = work_atom->next) {
  if ( work_atom->atomic_number != UINT_MAX) {
    /* We copy this one */
    new_molecule[new_molecule_index].label = work_atom->label;
    new_molecule[new_molecule_index].atomic_number
    = work_atom->atomic_number;
    new_molecule[new_molecule_index].coordinates
    = work_atom->coordinates;
    new_molecule[new_molecule_index].valence = work_atom->valence;

    /* This is a bit tricky, we can copy the bonds from the old */
    /* molecule, but we won't be able to align them until next pass */
    new_molecule[new_molecule_index].bond = work_atom->bond;
    new_molecule[new_molecule_index].bond_order = work_atom->bond_order;

    new_molecule[new_molecule_index].formal_charge
    = work_atom->formal_charge;

    /* Qcodes in the new fragment will no longer be valid, so */
    /* simply set them to null, and free the work atom qcode */
```

```
    new_molecule[new_molecule_index].qcode = NULL;
    if(work_atom->qcode) { free(work_atom->qcode); }
    new_molecule[new_molecule_index].charge = work_atom->charge;
    new_molecule[new_molecule_index].s_descriptor
    = work_atom->s_descriptor;
    new_molecule[new_molecule_index].other = work_atom->other;
    new_molecule[new_molecule_index].state = work_atom->state;

    new_molecule[new_molecule_index].next =
    new_molecule_index == ( new_size - 1 ) ?
    NULL :
    &(new_molecule[new_molecule_index + 1]);
    new_molecule[new_molecule_index].previous =
    new_molecule_index == 0 ?
    NULL :
    &(new_molecule[new_molecule_index - 1]);
    new_molecule_index++;
  } else {
    /* We free memory from, and discard this one */
    if (work_atom->label) { free(work_atom->label); }
    if (work_atom->coordinates) { free(work_atom->coordinates); }

    /* Again, we'll free the bonds on the next pass */
    if (work_atom->qcode) { free(work_atom->qcode); }

    /* And, we do nothing with the other pointer.  Applications */
    /* that use it are responsible for their own memory handling */
  }
  }

  /* Now, make another pass and straighten out the bonds */
  for (work_atom = old_molecule_base; work_atom; work_atom = work_atom->next) {
  int new_molecule_index = 0, i, j;
  if ( work_atom->atomic_number != UINT_MAX) {
    /* We fix the bonds on this one */
    for ( i = 0; i < work_atom->valence; i++ ) {
    /* If the new bond is already in the new molecule, skip it */
    if ( work_atom->bond[i] - get_atom_offset(work_atom->bond[i])
       == new_molecule ) { continue; }

    /* The real new_molecule partner has copied the addy of this */
    /* label for itself, we'll find that one (for each bond) */
    for ( j = 0; j < new_size; j++ ) {
      if ( work_atom->bond[i]->label == new_molecule[j].label ) {
      work_atom->bond[i] = &(new_molecule[j]);
      }
    }

    /* If the bond still points to the old_molecule, make it NULL */
    if ( work_atom->bond[i] - get_atom_offset(work_atom->bond[i])
       == old_molecule_base ) {
      work_atom->bond[i] = NULL;
    }
    }
    new_molecule_index++;
  } else {
    /* We ignore this one */
  }
  }

  /* And pack the molecule's bonds */
  molecule_pack_all_bonds(new_molecule);

  /* Finally, in one last pass, we free the old bond and bond order pointers */
  for (work_atom = old_molecule_base; work_atom; work_atom = work_atom->next) {
  if ( work_atom->atomic_number != UINT_MAX) {
    /* We can now safely ignore this one */
  } else {
    /* We free the bond and bond orders here */
    if ( work_atom->bond ) { free(work_atom->bond); }
    if ( work_atom->bond_order ) { free(work_atom->bond_order); }
  }
  }

  /* And free the whole old atom array */
  free(old_molecule_base);

  return new_molecule;
}

/* The next three functions are used to manipulate and access a binary */
/* number.  */
void mark2(long int *number, int place) {

  /* This only does well up to 30 bits with long ints */
  if(place > 30) {
    warn_out("mark2 asked to handle more than 30 bits");
  }
  if (place < 0) {
    warn_out("mark2 given negative place to mark, results may vary");
  }

  /* As far as I can tell, the << operator guarantees shifting in 0's from */
  /* the high end, but the >> operator guarantees nothing.  For portability, */
  /* then, I'll only use the left shift operator */
  *number = ( *number | (1 << place) );

  return;
}

void unmark2(long int *number, int place) {

  /* This only does well up to 30 bits with long ints */
  if(place > 30) {
    warn_out("unmark2 asked to handle more than 30 bits");
  }
  if (place < 0) {
    warn_out("unmark2 given negative place to mark, results may vary");
  }

  *number = ( *number & ( ~( 1 << place ) ) );
  return;
}

boolean check2(long int value, int place) {

  if(place > 30) {
    warn_out("Warning, check2 asked to handle more than 30 bits");
  }

  return ( value & ( 1 << place ) ) ? true : false;

}

/* The following function is meant to be called first (in main) as it
   has the responsibility of reading all of the coordinates, and
   allocating the memory for the entire molecule.  It is also responsible
   for formatting the next and previous links, as well as assigning the
   atomic numbers.  It returns a pointer to the base of the new
   molecule.  Note that this function will 'eat' either one more line than
   the end of the formatted coordinates, or up to the end of the input
   file.  There is no way to distinguish between these two cases.  In
   the future, it can be written (possibly) to read a line from the
   input stream without eating the line, but this is likely to use
   specific system calls and decrease portability.  */
atom *read_init_formatted_coordinates(FILE *read_stream) {

  atom *molecule_base = NULL;
  int molecule_size = 0, i;
  boolean do_loop = true;
  char work_string[MAXSTR], another_string[MAXSTR];
  double x, y, z;


  while( do_loop ) {

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  if ( (fgets(work_string, MAXSTR, read_stream) ) == NULL) { break; }

  /* Initialize a new atom variable, and fill in the appropriate values */
  if (sscanf(work_string, "%[^,],%lf,%lf,%lf\n",
         another_string, &x, &y, &z) == 4 ) {

#ifdef DMALLOC
    dmalloc_verify(0);
#endif

    /* We are reading coordinates */
    molecule_base = atom_realloc(molecule_base,
                  ++molecule_size,
                  strlen(another_string) + 1);

#ifdef DMALLOC
    dmalloc_verify(0);
#endif

    /* molecule_size is currently also the index of the last element */
    strcpy(molecule_base[molecule_size - 1].label, another_string);

#ifdef DMALLOC
    dmalloc_verify(0);
#endif

    molecule_base[molecule_size - 1].coordinates[0] = x;
    molecule_base[molecule_size - 1].coordinates[1] = y;
    molecule_base[molecule_size - 1].coordinates[2] = z;

#ifdef DMALLOC
    dmalloc_verify(0);
#endif

  } else {
    do_loop = false;
  }
  }

  if( molecule_size == 0 ) {
  error_exit("Could not read any coordinates from designated "
         "file stream in read_init_formatted_coordinates()");
  }

  /* Initialize array to contain the atomic numbers.  Also, while we're */
  /* looping through the array, we might as well configure the links as */
  /* well */

  for (i = 0; i < molecule_size; i++) {
  if ((molecule_base[i].atomic_number =
     atom_lab_to_num(molecule_base[i].label)) == 0) {
    warn_out("unknown atom type encountered in main");
  }
  if ( i != 0 ) {
    molecule_base[i].previous = molecule_base + i - 1;
  }
  if ( i != ( molecule_size - 1 ) ) {
    molecule_base[i].next = molecule_base + i + 1;
  }
  }

  return molecule_base;
}

/* The following fucntion reads formatted coordinates from the designated
   file stream.  It is designed to be called right after
   read_init_formatted_coordinates().  It may _not_ actually format all
   of the connectivity correctly, especially if the connectivity
   information provided isn't formatted _perfectly_ or is incompete.
   other functions will have the responsibility for verifying this
   information.  */

void read_formatted_connectivity(FILE *read_stream, atom *member) {

  atom *molecule_base;
  char work_string[MAXSTR];
  int i, j;
  float this_bond_order;
```

```
  boolean do_loop = true;

  if (!member) {
  error_exit("NULL atom pointer passed to read_formatted_coordinates()");
  }

  molecule_base = member - get_atom_offset(member);

  while(do_loop) {

  if( (fgets(work_string, MAXSTR, read_stream) ) == NULL) { break; }
  /* Read connectivity information */
  if (sscanf(work_string, "%d %d %g", &i, &j, &this_bond_order) == 3) {
    atom_connect(molecule_base + i, molecule_base + j, this_bond_order);
  } else {
    do_loop = false;
    break;
  }
  }

  return;
}

/* The following function very simply returns a pointer to the base of
   the molecule, where some_atom, is any atom in that molecule */

atom *molecule_return_base (atom *some_atom) {
  if (some_atom == NULL) {
  error_exit("Null pointer passed to molecule_return_base");
  }
  while (some_atom->previous) { some_atom = some_atom->previous; }
  return some_atom;
}

/* The following function normalizes the charges on the molecule to
   the charge provided by total charge.  It does this by simply adding
   the total charge / number of atoms to each charge in the molecule */

void molecule_normalize_charges(atom *member, float total_charge) {

  double charge_sum, charge_offset;
  atom *molecule_base;

  /*  printf("Total charge is %g\n", total_charge);
    printf("The molecule's size is %d\n", molecule_get_size(member));*/

  charge_sum = -total_charge;
  molecule_base = member = molecule_return_base(member);

  while ( member ) { /* (is non-null) */
  charge_sum += member->charge;
  member = member->next;
  }

  charge_offset = charge_sum /   molecule_get_size(molecule_base);
  member = molecule_base;

  while ( member ) {
  member->charge -= charge_offset;
  member = member->next;
  }

  return;
}

/* The following function generates a list of all unique angles in a
   molecule.  It allocates the space for the list.  The format of the
   list is that it's flat, with the sentry (end) values being 0 0 0.
   Each entry is a triplet, with the second atom being the central
   atom.  */
int *generate_angle_list(atom *some_atom) {

  int *angle_list, a_list_size, m_size, i, j, k;
  atom *molecule_base;

  if (some_atom == NULL) {
  warn_out("Null atom passed to generate_angle_list(), this will "
       "most likely be fatal");
  return NULL;
  }

  a_list_size = 0;
  angle_list = malloc( (a_list_size + 1) * 3 * sizeof(int *) );
  if(angle_list == NULL) {
  error_exit("Unable to allocate space for angle_list in "
         "generate_angle_list()");
  }
  angle_list[0] = 0;
  angle_list[1] = 0;
  angle_list[2] = 0;

  /* Now, we loop through all of the atoms, and add the angles as we
   find them */
  molecule_base = molecule_return_base(some_atom);
  m_size = molecule_get_size(some_atom);
  for ( i = 0; i < m_size; i++ ) {
  if( molecule_base[i].valence >=2 ) {
    for ( j = 0; j < molecule_base[i].valence - 1; j++ ) {
    for ( k = j + 1; k < molecule_base[i].valence; k++ ) {
      angle_list = realloc( angle_list,
              (a_list_size + 2) * 3 * sizeof(int *) );
      if (angle_list == NULL) {
      error_exit("Unable to reallocate angle_list in "
           "generate_angle_list()");
      }

      /* We have a new angle, record it */
      angle_list[3 * (a_list_size) + 0] =
      get_atom_offset(molecule_base[i].bond[j]);
      angle_list[3 * (a_list_size) + 1] = i;
      angle_list[3 * (a_list_size) + 2] =
      get_atom_offset(molecule_base[i].bond[k]);

      /* Reset the sentry values, and increment the size of the
       list */
      angle_list[3 * (a_list_size + 1) + 0] = 0;
      angle_list[3 * (a_list_size + 1) + 1] = 0;
      angle_list[3 * (a_list_size + 1) + 2] = 0;
      a_list_size++;
    }
    }
  }
  }
  return angle_list;
}
```

Vector Handling Library

vector.h

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* Includes */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* The following is a memory debugging library */
#ifdef DMALLOC
#include <dmalloc.h>
#endif

/* Defines */
#define VECDVALSIZE sizeof(double)

/* Typedefs and structs */

typedef double *vector_d;
typedef vector_d* matrix_d;

/* Functions in vector_handling.c */

/* Functions that allocate new memory */
vector_d vec3_cross_product(vector_d vec1, vector_d vec2);
vector_d vec_orthogonalize(vector_d norm_base, vector_d orth_part,
  int dimension);
/* vec_proj calls vec_scalar_multiply */
vector_d vec_proj(vector_d vec_base, vector_d vec_to_proj, int dimension);
vector_d vec_scalar_multiply(double scalar, vector_d vector, int dimension);
vector_d vec_subtract(vector_d subtract_from, vector_d subtract_vector,
  int dimension);
vector_d vec_add(vector_d vector1, vector_d vector2, int dimension);
vector_d vec_normalize(vector_d vector1, int dimension);
vector_d vec_transform(matrix_d orthonormal_transform_matrix,
  vector_d vector_to_transform, int dimension);

/* Functions that do not allocate new memory */
double vec_dot(vector_d vector1, vector_d vector2, int dimension);
double vec_scalar_length(vector_d a_vector, int dimension);
void vec_print(vector_d some_vector, int dimension);
```

vector_handling.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
```

```
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include "vector.h"

/* Typedefs */

/* Warning: this function allocates memory */
vector_d vec3_cross_product(vector_d vector1, vector_d vector2) {

  vector_d work_vector;

  /* Allocate space for work vector */
  if ((work_vector = malloc(3 * sizeof(double))) == NULL) {
  fprintf(stderr,
  "Error allocating memory for work_vector in vec3_cross_product.\n");
  exit(0);
  }
  work_vector[0] = vector1[1] * vector2[2] - vector1[2] * vector2[1];
  work_vector[1] = vector1[2] * vector2[0] - vector1[0] * vector2[2];
  work_vector[2] = vector1[0] * vector2[1] - vector1[1] * vector2[0];

  return work_vector;
}

/* The following function returns the portion of vector2 that is */
/* normal to vector1.  Dim referrs to the dimension, as this routine */
/* is easily built to handle any dimensional space */

/* Warning:  This function allocates memory for a new vector */

vector_d vec_orthogonalize(vector_d vector1, vector_d vector2, int dim) {

  int i;
  int vecok1 = 0, vecok2 = 0;
  vector_d work_vector, work_vector2;

  /* Check for null vectors */
  for (i = 0; i < dim; i++) {
    if (vector1[i] != 0) vecok1 = 1;
    if (vector2[i] != 0) vecok2 = 1;
  }
  if ( (vecok1 * vecok2) == 0 ) {
    fprintf(stderr,
      "Warning:  null vector passed to vec_orthogonalize\n");
    return NULL;
  }

  /* Now, the portion of vector2 perpendicular to vector 1 is given */
  /* by: vector2 - proj(vector1)vector2.  And proj(vector1)vector2 */
  /* is given by (vector1 dot vector2) / (len vector2 ^2) *       */
  /* (vector1) */

  work_vector = vec_proj(vector1, vector2, dim);
  work_vector2 = vec_subtract(vector2, work_vector, dim);

  /* It's always good to clean house, the memory at work_vector */
  /* will be lost after function exit */
  free(work_vector);

  return work_vector2;
}

/* This function takes 2 vectors and a dimension as arguments and */
/* returns the projecton of vector2 onto vector1.  This quantity is */
/* defined: proj(vector1)vector2 is given by (vector1 dot vector2) / */
/* (len vector2 ^ 2) * (vector1) */

/* Warning: this function allocates new memory via it's call to */
/* vec_scalar_multiply */
vector_d vec_proj(vector_d vector1, vector_d vector2, int dim) {

  vector_d work_vector;
  float x, y;

  x = vec_dot(vector1, vector2, dim);
  y = vec_scalar_length(vector1, dim);
  y = y * y;

  work_vector = vec_scalar_multiply((x/y), vector1, dim);

  return work_vector;
}

/* This function takes 2 vectors and a dimension and returns */
/* vector1 dot vector2 */

double vec_dot(vector_d vector1, vector_d vector2, int dim) {

  double sum = 0;
  int i;

  for (i = 0; i < dim; i++) {
    sum += vector1[i] * vector2[i];
  }

  return sum;
}

double vec_scalar_length(vector_d vector, int dim) {

  int i;
  double x = 0;

  for (i = 0; i < dim; i ++) {
    x += vector[i] * vector[i];
  }

  x = sqrt(x);

  return x;
}

/* The following function simply returns the resultant vector of */
/* a scalar multiplication */

/* Warning:  This function allocates memory for a new vector */

vector_d vec_scalar_multiply(double scalar, vector_d vector, int dim) {

  int i;
  vector_d work_vector;

  /* Allocate space for work vector */
  if ((work_vector = malloc(dim * sizeof(double))) == NULL) {
  fprintf(stderr,
  "Error allocating memory for work_vector in vec_scalar_multiply.\n");
  exit(0);
  }

  for (i = 0; i < dim; i++) {
    work_vector[i] = scalar * vector[i];
  }

  return work_vector;
}

/* This function returns a vector which is equal to vector1 - vector2 */

/* Warning:  This function allocates memory for a new vector */

vector_d vec_subtract(vector_d vector1, vector_d vector2, int dim) {

  int i;
  vector_d work_vector;

  /* Allocate space for work vector */
  if ((work_vector = malloc(dim * sizeof(double))) == NULL) {
  fprintf(stderr,
  "Error allocating memory for work_vector in vec_subtract.\n");
  exit(0);
  }

  for (i = 0; i < dim; i++) {
    work_vector[i] = vector1[i] - vector2[i];
  }

  return work_vector;
}

/* This function returns a vector which is equal to vector1 + vector2 */

/* Warning:  This function allocates memory for a new vector */

vector_d vec_add(vector_d vector1, vector_d vector2, int dim) {

  int i;
  vector_d work_vector;

  /* Allocate space for work vector */
  if ((work_vector = malloc(dim * sizeof(double))) == NULL) {
  fprintf(stderr,
  "Error allocating memory for work_vector in vec_add.\n");
  exit(0);
  }

  for (i = 0; i < dim; i++) {
    work_vector[i] = vector1[i] + vector2[i];
  }

  return work_vector;
}

void vec_print(vector_d vector, int dim) {

  int i;

  for (i = 0; i < dim; i++) {
    if (i != (dim - 1)) {printf("%f,", vector[i]);}
    else {printf("%f", vector[i]);}
  }
/*  printf(" %f\n",vec_scalar_length(vector, dim)); */
  printf("\n");

}

/* Warning:  This function allocates memory for a new vector */

vector_d vec_normalize(vector_d vector, int dim) {

  vector_d work_vector;
  double norm_value;

  norm_value = vec_scalar_length(vector, dim);
  work_vector = vec_scalar_multiply( (1 / norm_value), vector, dim);

  return work_vector;
}

/* This function transforms vector from normal cartesian to transform */
/* system.  Note that in order for this transformation to make sense, */
/* transform _must_ be orthonormalized */

/* Warning:  This function allocates memory for a new vector */

vector_d vec_transform(matrix_d transform, vector_d vector, int dim) {

  vector_d work_vector;
  int i;

  /* Allocate space for work vector */
  if ((work_vector = malloc(3 * sizeof(double))) == NULL) {
  fprintf(stderr,
  "Error allocating memory for work_vector in vec_normalize.\n");
  exit(0);
  }
```

```
  for (i = 0; i < dim; i++) {
    work_vector[i] = vec_dot(vector, transform[i], dim);
  }

  return work_vector;
}
```

Miscellaneous

chkmem.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* This program takes a single argument (an int), and tries to malloc */
/* that many megabytes of ram.  If it fails, it returns 0, if it */
/* succeeds, it returns 1.  Pretty simple, really.  Note that it both */
/* returns the required value, and prints it as it's output.  It does */
/* little error chekcing, and is meant to be used only by external programs. */
/* When compiled, it should be compiled to a program named chkmem */

#include <stdlib.h>

int main(int argc, char* argv[]) {

  int *chk = NULL, total;

  if (argc != 2) {
    putchar('0');
    return 0;
  }

  total = strtol(argv[1], NULL, 10) * 1024 * 1024;

  chk = malloc(total);

  if (chk != NULL) {
    free(chk);
    putchar('1');
    return 1;
  } else {
    putchar('0');
    return 0;
  }
}
```

my_socket.h

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>

/* The following is a memory debugging library */
#ifdef DMALLOC
#include <dmalloc.h>
#endif

/* Typedefs */
#ifndef BOOLEAN
#define BOOLEAN
typedef enum {false = 0, no = 0, true = 1, yes = 1} boolean;
#endif

/* Defines */
#ifndef MAXSTR
#define MAXSTR 256
#endif

/* Prototypes in my_socket.c */
int get_new_socket(const char* hostname, const unsigned int port);
boolean sendall(int socket_descriptor, char *to_send, int *length);
void socket_finish_send(int socket_descriptor);
char *recvall(int socket_descriptor);

/* Prototypes necessary but not '#include' 'ed */
void error_exit(char *message);
```

my_socket.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include "my_socket.h"

int get_new_socket(const char* hostname, const unsigned int port) {

int sockfd;
struct sockaddr_in remote_sock;
struct hostent *h;

if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) == -1) {
  printf("Unable to get sock, exiting\n");
  perror("socket");
  exit(0);
}

if ( (h = gethostbyname(hostname)) == NULL ) {
  printf("Unable to lookup ip address");
  herror("gethostbyname");
  exit(0);
}

/* This is how we would see what we got */
/* printf("Host name  : %s\n", h->h_name); */
/* printf("IP Address : %s\n", inet_ntoa(*((struct in_addr *)h->h_addr))); */

remote_sock.sin_family = AF_INET;
remote_sock.sin_port = htons(port);
remote_sock.sin_addr = *((struct in_addr *)h->h_addr);
memset(remote_sock.sin_zero, '\0', 8);

/* This section was written earlier, as I would sometimes get */
/* 255.255.255.255 for an address.  This is _always_ an invalid */
/* address (for our purposes) so I chose to leave this section in */
if (remote_sock.sin_addr.s_addr == -1) {
  /* We failed to get an ip address, return an error */
  printf("We got 255.255.255.255 for an address, this is an error, "
      "exiting \n");
  exit(0);
}

if (connect(sockfd, (struct sockaddr *)&remote_sock,
  sizeof(struct sockaddr)) == -1 ) {
  printf("Could not connect for some reason\n");
  perror("connect");
  exit(0);
}

return sockfd;
}

/* Note:  This function, and much of the basis for other socket handling */
/* came from an excellent tutorial on network programming.  It can be found */
/* at:  http://www.ecst.csuchico.edu/~beej/guide/net/ */

boolean sendall(int socket_descriptor, char *to_send, int *length) {

/* The variables declared have the following meanings: */
/* total        How many bytes we've sent */
/* bytesleft    How many left to send.  Note:  This changes the value */
/*              passed from the calling function, if it's of interest */
```

```
int total = 0;
int bytesleft;
int n;

bytesleft = *length;

while (total < *length) {
  n = send(socket_descriptor, to_send + total, bytesleft, 0);
  if (n == -1) {
    printf("Some form of local error encountered by send() in "
         "sendall().  Returning false");
    perror("send");
    break;
  }
  total += n;
  bytesleft -= n;
}

*length = total;
/* Sets length to the length actually sent */

if (n == -1) {
  /* Failure */
  return false;
} else {
  /* Success */
  return true;
}

/* End of function */
}

/* The following function receives data of up to MAXSTR - 1 characters, */
/* terminated by crlf, and places it in the return value (as a static). */
/* This value must be stored before calling this function again */
char *recvall(int socket_descriptor) {

static char return_string[MAXSTR * 4], crlf[3];
char recv_buffer[MAXSTR * 4];
static int initialized = 0;
int bytes_received, space_left;
char *buffer_position, *temp_p;
boolean continue_receiving = yes;

#ifdef DMALLOC
dmalloc_verify(0);
#endif

/* Initialize crlf */
if (!initialized) {
  initialized = 1;
  crlf[0] = '\15';
  crlf[1] = '\12';
  crlf[2] = '\0';
}

buffer_position = recv_buffer;
recv_buffer[0] = '\0';
recv_buffer[MAXSTR * 4] = '\0';
space_left = MAXSTR * 4;
while (continue_receiving && space_left > 1) {
  bytes_received =
    recv(socket_descriptor, buffer_position, space_left - 1, 0);
  /* Append a null character to make the string processable */
  *(buffer_position + bytes_received) = '\0';

  if (bytes_received == 0) {
    error_exit("Received 0 bytes on blocking call to recv() in "
          "recvall().  This is an unknown error condition.  "
          "(Server hung up?)");
  }

  /* Decrement the amount of space left */
  space_left -= bytes_received;

  if( (temp_p = strstr(buffer_position, crlf) ) ) {
    /* We've got the whole string, change the last character and */
    /* move on out */
    continue_receiving = no;
    *(temp_p) = '\0';
  } else {
    /* Move up the buffer position */
    buffer_position += bytes_received;
  }
}

#ifdef DMALLOC
dmalloc_verify(0);
#endif

if (space_left == 1) {
  /* We filled the buffer.  Simply return NULL */
  return NULL;
}

strcpy(return_string, recv_buffer);

return return_string;

}

/* The following function simply sends the end of line combination, */
/* confirming the send */

void socket_finish_send(int socket_descriptor) {

int n;
static int initialized = 0;
static char crlf[3];

/* Initialize crlf */
if (!initialized) {
  initialized=1;
  crlf[0] = '\15';
  crlf[1] = '\12';
  crlf[2] = '\0';
}

n = send(socket_descriptor, crlf, 2, 0);
if (n == -1 || n != 2) {
  printf("Some form of local error encountered by send() in"
    " sendall().  Returning false");
  perror("send");
}

return;
}
```

total_atom_byte_size.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include <stdio.h>
#include <stdlib.h>
#include "atom.h"

int main () {

int total;

total = sizeof(atom);
total += 3 * sizeof(char);
total += 4 * sizeof(atom *);
total += 4 * sizeof(float);
total += 20 * sizeof(double);


printf("The size of an atom (total) is %d\n", total);

return 0;
}
```

general/os_specific

generic_make.pl

```
# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This is the 'generic' config file.  Unless there's an obvious change
# to how the various ./configure.pl's have been written, it's much better
# to either copy this to your own os's configuration file, or copy one of
# the other configuration files

# The following line closes the 'Preparing makefile for $os' line
print "\n";

# Set any variables in the next section - see commented lines for examples
# $main::cflags .= " -Wall";
# $main::cc = "gcc";

# $main::profile = "-p";
```

```
# If there's any conditional actions to be done, do them here.



# And ... make this library return true
1;
```

linux-make.pl

```
# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This file contains code that needs to be placed in configure.pl for
# linux-gnu.  It can be modified for your particular os.

print "The author had access to this system\n" .
  "during development\n";

$main::cflags .= " -Wall";
$main::cc = "gcc";

# Since we're ocassionally using functions that are system specific (as to
# their locations), we need to specify that here.  This supresses the
# compiler warnings 'implicit declarations of function ...'
$main::defines .= " -D_GNU_SOURCE";

$main::profile = "-pg";

# If we're on a ppc (at least for the G4 that the author did some development
# on) we need to use a less aggressive debugging flag.

my($machine_hardware) = `uname -m`;
chomp($machine_hardware);

if ($machine_hardware =~ /^([-\w]+)$/) {
  $machine_hardware = $1;  #$machine_hardware is laundered
} else {
  die "Bad hardware \"$machine_hardware\" retrived from " .
  "environment, exiting";
}

if ( $machine_hardware eq "ppc" ) {
  $main::debug = "-g";
} else {
  $main::debug = "-ggdb3";
}

# And ... make this library return true
1;
```

dec_osf_make.pl

```
# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This file contains configuration for the osf4.0f os.  A significant
# portion of the development was done on this type of machine.

print "The author had access to this system during development\n";

# The longdoubleny1 warning concerns the fact that on the DECs (jubjub
# and company) there is no difference between double and long double.
# The "msg_disable warninplicitfunc" referrs to the fact that with
# warnings disabled, the compiler complains if we don't have prototypes
# for things like <stdio.h>.  One would think that simply #include 'ing
# these header files would be sufficient, but this is apparently not
# acceptable.  I've dug into this problem a bit, but have not yet
# found an acceptable answer to this problem

$main::cflags .= " -verbose -w2 -warnprotos -msg_disable longdoubleny1," .
  "warnimplfunc";

$main::profile="-gen_feedback -prof_gen";


# And ... make this library return true
1;
```

genff

.ff_form

```
# This file provides the form of the force field that we want to use.
# All force files will rely on the simple fact that the total energy
# (and force) are dependent on a simple sum of energy terms.  The
# terms are listed on uncommented lines, and refer to energy
# evaluation functions that must be compiled into the working energy
# evaluation code, or at least available on the system as a shared
# library.  For initial development, all functions will simply be
# compiled statically into the final executable.  Any options that the
# functions themselves take must be included in parentheses following
# the function.  These will be parsed, and the original code will
# support nesting of parentheses (initially, the parser will only
# handle two levels of parenthesis.  If the need arises later for more
# flexibility, it will be added).  One 'special' function is the
# global() function.  It's purpose is to provide any global parameters
# the simulation will need.  Note that nothing should be put into the
# global structure that really belongs in an individual function.
# Further information on the global information structure can be found
# in nrgforce.h.  The energy evaulation functions themselves are
# collected in nrgforce.c.  Any information _after_ a line beginning
# with END will be ingnored, and need not be commented.  Other
# limitations: The function or keyword (i.e., global), must be at the
# beginning of the line.  Note that extended information (within the
# parentheses, for example) may span multiple lines.  Options provided
# within parentheses will be parsed by either commas or spaces.  If
# you desire several items to be within the first field (which is
# flags for the function), provide them in another nested level of
# parentheses.  Finally, a function request must be terminated by a
# semicolon.

global();
bond_gen_dreiding2();
bend_gen_dreiding2();
torsion_bouldergroup1();
inv_gen_dreiding2();
vdw_bouldergroup1(OMIT12, OMIT13, OMIT14);
coul_rawsum(OMIT12, OMIT13, OMIT14);

END force filed definitions
^^^

Do _not_ delete this end, or the rest of the fill will be interpreted
into your force field!

This section describes (and documents briefly) all available energy
evaluation functions.  If a new energy evaluation function is
written, it should be documented here.  Additional documentation can
be found at the beginning of the actual function, found in
nrgforce.c.  Note that this the 'most current' version of this can be
found at the end of nrgforce.h.  If we create a 'configuration file
builder', it may end up reading the end of the header file, and
including this information there.

global()
options: none

bond_gen_dreiding2()
options: none

bend_gen_dreiding2()
  options: none

torsion_bouldergroup_1()
  options: none

inv_gen_dreiding2()
  options: none

vdw_bouldergroup_1(OMIT12, OMIT13, OMIT14)
  options:  OMIT12  Omit 1,2 non-bonded interactions
        OMIT13  Omit 1,3 non-bonded interactions
        OMIT14  Omit 1,4 non-bonded interactions

coul_rawsum(OMIT12, OMIT13, OMIT14)
  options:  OMIT12  Omit 1,2 non-bonded interactions
        OMIT13  Omit 1,3 non-bonded interactions
        OMIT14  Omit 1,4 non-bonded interactions
```

get_torsion_parameters.h

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include <stdio.h>
#include <stdlib.h>
#include "../sim/nrgforce.h"
/* We do not include ../general/atom.h here, since ../sim/nrgforce.h
   already includes that file */

/* The following is a memory debugging library */
#ifdef DMALLOC
#include <dmalloc.h>
#endif

#ifndef BOOLEAN
#define BOOLEAN
typedef enum {false = 0, no = 0, true = 1, yes = 1} boolean;
#endif

/* Define a function pointer --- may be useful later?
   double (*fcn_ptr) (atom**, long, double*, global_sim_parms);
*/

/* Functions defined at the end of get_torsion_parameters.c */
atom *read_init_qdbis_input(FILE *stream);
void initialize_charges_from_file(atom *molecule, char* filename,
                  int total_charge);
```

get_torsion_parameters.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include "get_torsion_parameters.h"

#ifdef DMALLOC
  #include <dmalloc.h>
#endif

/* This program takes as input a processed file from
   qdb_input_server.pl, and outputs a finished force field.  It
   receives the file on standard in, prints the finished force field
   to standard out, and any warning or errors are output to standard
   error. */

/* This is the program for testing the dynamic building of force
   fields */

int main(int argc,char *argv[]) {

  atom *molecule;
  atom *sim_sys[2];

  /* Initialization of command line variables */
  if ( argc != 2 ) {
  printf(
       /* Begin usage message */
       "Usage:\n"
       "\n"
       "%s <charges_input_file>\n"
       "\n"
       "Where <charges_input_file> is the file that has the (already\n"
       "calculated) charges for the molecule whose structure has been\n"
       "provided on stdin.\n"
       "\n"
       "This program is meant to be called by a master program",
       argv[0] );
  error_exit("");
  }

  /* The first task is to read the input geometry, and initialize the
   molecule (using the atom_handling module)  */

  molecule = read_init_qdbis_input(stdin);

  if ( molecule == NULL ) {
  error_exit("Unable to initialize input molecule");
  } else {
  fprintf(stderr, "Initialization of molecule from input successful\n");
  }

  /* The basic input is finished, but we'll still need to initialize
   the charges.  The proper filename has been provided on the
   command line.  */


  initialize_charges_from_file(molecule, argv[1], 0);

  /* And finally, we need to initialize our simulation system, which
   is simply one molecule.  The energy handling functions, however,
   expect a list of molecules, so we provide that here */
  sim_sys[0] = molecule;
  sim_sys[1] = NULL;

  /* print_molecule(stdout, molecule); */

  /* The initialization is complete.  Note that the user of the
   nrgforce.c functions never actually touch the force field, nor
   do they do any updating of the molecule they have.  They can
   request the library return energies for a system's configuration,
   and they can request that forces or positions be updated.  This
   allows simulation code to be written completely separately from
   the actual implementation of the evaluations. */

  /* Define for ourselves a function pointer to use while resolving
   exactly how to create a run time force field */

  if (init_ff(".ff_form", sim_sys)) {
  fprintf(stderr, "Initialization successful\n");
  } else {
  error_exit("Initialization failed\n");
  }

  /* The next leg of development is significantly trickier than
   probably anything done up to this point in development.  The
   general outline of what to do follows:

   1) Be able to evaluate the energy of a given conformation
   2) Be able to manipulate the conformation of the molecule with
      simple to use functions.
   3) Allow the user to 'rig' the torsion_bouldergroup1 function to
      return either 0 energy contribution, or to specifically set
      the form of the fourier series.
   4) Write functions (for this program) to read geometries of
      database fragments, and map the geometries onto our parent.
  */

#if 1
  {
  /* Testing block */
  molecule->coordinates[0] = 0.01;
  printf("\nThe energy of the initial system is %Lg\n\n",
       get_system_energy(sim_sys, (int)RETURN_ENERGY));
  }
#endif


  fprintf(stderr, "get_torsion_parameters exiting.\n");

  /* Before exiting, free all used memory (so we can catch leaks when
   they happen */

  /* TODO:  I seem to have some memory that's not associated with a
   particular function (probably a system function?) that's not
   being allocated.  Check into this with a debugger at some point. */

  free_molecule(molecule);

#ifdef DMALLOC
    dmalloc_shutdown();
#endif

  return 0;
}


/* This function reads an input atom from a qdb_input_server.pl
   processing run, and initializes the molecule.  It will take
   advantage of code already provided in ../qdb/qdb_check_functions.c
   for several of its steps, simply discarding expected lines as we
   see them. */
atom *read_init_qdbis_input(FILE *stream) {

  char line[MAX_FLINE], *string_p;
  atom *molecule = NULL;

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  /* Grab first two lines of input, and verify that they're correct */
  if ( (fgets(line, MAXSTR, stream) ) == NULL) { return NULL; }
  string_p = "Begin parent molecule:";
  if ( strncmp( line, string_p, strlen(string_p) ) ) {
  printf("Failed to match input string: %s\nto expected string:"
       "%s\n\t in read_init_qdbis_input() returning ", line, string_p);
  return NULL;
  }
  if ( (fgets(line, MAXSTR, stream) ) == NULL) { return NULL; }
  string_p = "Begin coordinates";
  if ( strncmp( line, string_p, strlen(string_p) ) ) {
```

```c
printf("Failed to match input string: %s\nto expected string:"
     "%s\n\t in read_init_qdbis_input() returning ", line, string_p);
return NULL;
}

molecule = read_init_formatted_coordinates(stream);

/* We cannot check the next line to make certain the input is
 correct, as read_init_formatted_coordinates() eats one more line
 than it needs.  See the comments at the header of the function for
 ideas on re-writing this functionality Now, read connectivity
 information, and initialize it as we read it.  Since the
 information ultimately came from ../qdb/qdb_check, we will lend
 it some element of trust, and error checking need not be quite as
 thorough as we used there */

read_formatted_connectivity(stream, molecule);

/* Here is where we read the qcode information. There will be no
 verification of the qcodes done.  In order to do this correctly,
 we need to take ../qdb/assign_qcodes, and some other (scattered)
 functions, and define a qcode library in the general directory.
 For expedience, this is left as a future improvement.  Also, note
 that the next section is defined in its own block, to confine
 it's variables. */

{
atom *this_atom = molecule;
int qcode_pos;
long double *this_qcode;
boolean do_loop = true;

/* The following loop exits only on a break statement */
while ( 1 ) {
  if ( (fgets(line, MAX_FLINE, stream) ) == NULL) {
  warn_out("End of file reached before all qcodes were initialized, "
       "this will most likely be fatal");
  return NULL;
  }

  string_p = "Begin stereochemical descriptors";
  if ( !strncmp( line, string_p, strlen(string_p) ) ) {do_loop = false;}
  string_p = "End molecule output";
  if ( !strncmp( line, string_p, strlen(string_p) ) ) {do_loop = false;}
  if (do_loop == false) { break; }

  /* We're reading qcodes, parse away.  Note that we put no
   arbitrary limit on how long the qcodes can be */
  qcode_pos = 0;
  this_qcode = NULL;

  /* Split line up by spaces, and go */
  strtok(line, " ");
  while ( ( string_p = strtok(NULL, " ") ) ) {

  /* Expand the size of this_qcode */
  if ( ( this_qcode =
       realloc(this_qcode, (qcode_pos + 1) * sizeof(long double) ) )
     == NULL ) {
    error_exit("Unable to reallocate memory while initializing "
         "qcodes");
  }

  /* Copy the new value into the proper qcode space */

  /* An important note here:  While the function strtold is defined
     in the C99 standard, it is not included in the default libraries
     for osf4.0f.  We'll have to use a replacement function */
  /* this_qcode[qcode_pos] = strtold(string_p, NULL); */
  sscanf(string_p, "%Lf", &this_qcode[qcode_pos]);

  /* Increment position and keep going */
  qcode_pos++;
  }

  /* We've parsed the qcode, now to simply hand it off to the
   current atom.  (We do not free it, it belongs to the atom
   now).  Also, increment the atom.  */
  this_atom->qcode = this_qcode;
  this_atom++;
}
}

/* Here is where we read any stereochemical descriptors that may or
 may not exist */

string_p = "Begin stereochemical descriptors";
if ( !strncmp( line, string_p, strlen(string_p) ) ) {

int atom_number;
char descriptor;

while (1) {

  /* First, set our exit condition */
  if ( (fgets(line, MAXSTR, stream) ) == NULL) { return NULL; }
  string_p = "End molecule output";
  if ( !strncmp( line, string_p, strlen(string_p) ) ) { break; }

  if ( sscanf(line, "%i %1c", &atom_number, &descriptor) != 2 ) {
  /* This is an error */
  return NULL;
  }

  /* Initilalize! */
  molecule[atom_number].s_descriptor = descriptor;
}
}

/* Finally, this line must be the end of the input, or our input is
 hosed. */

string_p = "End molecule output";
if ( strncmp( line, string_p, strlen(string_p) ) ) {
  printf("Failed to match input string: %s\nto expected string:"
     "%s\n\t in read_init_qdbis_input() returning ", line, string_p);
  return NULL;
  }

  return molecule;
}

/* The following function reads charges and maps them directly to the
   molecule provided.  It does no error checking, but does normalize
   the charge.  I left the normalization till here, in case there is
   need to further modify the charge normalization scheme.  This
   function does no symmetrization of charge.  */

void initialize_charges_from_file(atom *this_atom, char* filename,
                 int total_charge) {

  FILE *charges_file;
  char line[MAXSTR];
  double this_charge;
  atom *molecule_base;

  molecule_base = this_atom = molecule_return_base(this_atom);

  if ( ! (charges_file = fopen(filename, "r") ) ) {
  error_exit("Unable to open charges file in "
        "initialize_charges_from_file()");
  }

  /* This is an easy job, simply read the charges line by line, and
   map them onto the molecule */

  while ( fgets(line, MAXSTR, charges_file) ) {
  if ( sscanf(line, "%lf", &this_charge) != 1 ) {

    /* There's some problem with the input file */

    error_exit("Reached unreadable line (charge) in input file "
         "before charges have been assigned to all of the "
         "atoms, it is likely that the charges file provided "
         "is not for the molecule supplied on stdin");
  }

  /* Set the value, and move on. */
  this_atom->charge = this_charge;
  if ( this_atom->next != NULL ) {
    this_atom++;
  } else {
    /* This is a bit tricky, we need to try to read another line,
     and also to try to convert it to a long float, if both
     succeed, then we probably don't have the correct input file */

    if ( fgets(line, MAXSTR, charges_file) &&
       sscanf(line, "%lf", &this_charge) == 1 ) {
    error_exit("Reached last atom before end of charges file, it "
          "is likely that the charges file provided is not "
          "for the molecule supplied on stdin");
    } else {
    /* Everything is in order, clean up and return */
    fclose(charges_file);

    /* Before returning, we need to normalize the charges for this
       molecule */
    molecule_normalize_charges(molecule_base, (double)total_charge);

    return;
    }
  }
  }

  /* If we ever get to the outside of the loop, there is a logical
   error in this function.  All exit conditions should be handled in
   the while loop */

  error_exit("Unknown logic error in initialize_charges_from_file()");

  return;

}
```

configure.pl

```perl
#!/usr/bin/perl -wT

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

package main;

eval { require 5.6.1 }
```

```
  or die <<MESSAGE;
###################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
###################################################################
MESSAGE

use strict;

# Before proceeding, clean up our environment so we can run external
# programs
require '../general/clean_environment.pl';
full_env_clean();

# This script will configure the makefile for whatever operating system
# we are running on.  It gets this information from the environment
# variable $OSTYPE, which will be laundered and used immediately as
# if it was secure.  I cannot perceive of any circumstances that would
# make this a security problem.

my($os)=$^O;

if ($os =~ /^([-\@\w.]+)$/) {
  $os = $1;  #$os is laundered
} else {
  die "Bad system \"$os\" retrived from environment, exiting";
}

my($path_to_root) = '../';

open ("MAKEFILE", ">./Makefile") or die
  "Unable to open makefile for writing, exiting";

print MAKEFILE <<MY_MESSAGE
#######################################################################
# This makefile was created by configure.pl.  Any changes to this
# makefile will be overwritten!  Please edit the configure.pl script
# instead.
#
#
MY_MESSAGE
;

# Values for files in this package, these are the mostly likely to
# change regularly, so they're put first
my($vpath) = "../general:../log2str:../qdb:../sim";
my($mainobjs) = "get_torsion_parameters.o";
my($headers) = "\tnrgforce.h atom.h vector.h \\\n" .
        "\t\tqdb_shared_functions.h get_torsion_parameters.h";
my($otherobjs) = "\tnrgforce.o vector_handling.o\\\n" .
        "\t\tget_bond_order.o atom_handling.o \\\n" .
        "\t\tqdb_shared_functions.o\n";
my($exefile) = "get_torsion_parameters";
my($exesources) = "get_torsion_parameters.c";
my($miscdeps)=<<MORESTUFF
vector_handling.o: ../general/vector.h
atom_handling.o: ../general/atom.h get_bond_order.o vector_handling.o\\
  qdb_shared_functions.o
nrgforce.o: nrgforce.h
get_torsion_parameters.c: get_torsion_parameters.h
MORESTUFF
;

# Default makefile values

# Note:  Just a reminder.  my($variable) variables are local only to this
# scope, which is main.  If another file is 'require' 'ed, it will _not_
# have access to these variables unless they're fully qualified (since
# I'm using strict vars).  Any variables that are changed in the os
# specific instructions must be scoped to main here.

$main::debug = "-g";
$main::profile = "";
$main::base = "";
$main::cc = "cc";
$main::cflags = '$(DEBUG) $(PROFILE)';
$main::libs = "-lm";
$main::incls = "";
$main::defines = "";
my($dmalloc) = "";
my($exeshellargs) = "";
my($nowarn) = "";

# Include any information for system specific options
my ($sys_config_file) = $path_to_root . 'general/os_specific/' .
  $os . '_make.pl';

if ( -r $sys_config_file ) {
  # Announce our finding.  Note, the file itself should provide the
  # newline, and any other additional information to print.
  print "Preparing makefile for $os.  ";

  # And use it
  require $sys_config_file;
} else {
  print "Don't know how to make makefile for $os (i.e., file\n" .
    "\"general/os_specific/${os}_make.pl\" not found).  Using \n" .
    "generic configuration file\n";
  require $path_to_root . 'general/os_specific/generic_make.pl';
}

# I haven't worked out anything too fancy for command line handling, but
# for now, I'll add the options in hodgepodge, and document as I go.  Note
# That command line processing doesn't happen until _after_ the defaults
# are entered.  This allows any of the defaults to be overwritten.  If any
# arguments that normally take a string are left empty, they will also
# be empty in the resulting Makefile.

# Special variables used only in dealing with command line options:
my($addcflags);
my($addlibs);
my($addincls);
my($adddefines);
my($help_flag);
my($profile_on);

my($all_options) = q/
  "debug|d:s"   =>  \$main::debug,
  "profile|p:s" =>  \$main::profile,     # Ignored unless -P is also provided
  "P"           =>  \$profile_on,  # Set this to turn profiling on for the
                                   # compiled executable.  Several default
                                   # profiling setups are provided by this
                                   # script.
  "cc|c=s"      =>  \$main::cc,          # Name of your compiler (overwrites)
  "--cflags=s"  =>  \$addcflags,   # Compiler options
  "--libs=s"    =>  \$addlibs,     # Additional libraries to link
  "--incls=s"   =>  \$addincls,    # Additional includes
  "--defines=s" =>  \$adddefines,  # Additional defines
  "dmalloc|D"   =>  \$dmalloc,     # User _must_ have dmalloc libraries
                                   # installed! (for memory debugging)
  "nowarn|N"    =>  \$nowarn,      # Turn off setup messages for using
                                   # dmalloc libraries
  "help|h|?"    =>  \$help_flag,
/;

use Getopt::Long;
Getopt::Long::Configure( qw/no_ignore_case_always bundling/ );
my($cmd_line) = GetOptions ( eval($all_options) );

# First order of business is to see if help was requested.  If so, simply
# print out how Getopt::Long was called.  This isn't beautiful, but it
# requires the least maintenence while options are added.
if ($help_flag) {
  print "Outputting Getopt::Long configuration.  \nSee " .
      "http://www.perldoc.com/perl5.6/lib/Getopt/Long.html for " .
      "more information.\n";
  print $all_options;
  print <<HELP_MSG

Notes:
  For options that take an optional string (as indicated by :s after the
  option name), omitting that string makes it empty in the resulting
  Makefile.  Providing the string overwrites the values provided by this
  script.

  For options that take a manditory string, the default behavior is to
  append that string to the existing value.

Makefile not written.
HELP_MSG
;
  exit 1;
}

# Handle profiling if requested
unless ($profile_on) {
  $main::profile = "";
}

if($addcflags) { $main::cflags .= " $addcflags"; }
if($addlibs) { $main::libs .= " $addlibs"; }
if($addincls) { $main::incls .= " $addincls"; }
if($adddefines) { $main::defines .= " $adddefines"; }

# If we have chosen to use the dmalloc library, do all the variable
# mashing here:
if ($dmalloc) {
  $main::libs .= " -ldmalloc";
  $main::defines .= " -DDMALLOC -DDMALLOC_FUNC_CHECK";
  $exeshellargs = "\t\@dmalloc -b -l dmalloc.txt -i 2 high -p " .
    "check-funcs -p check-heap -p log-blocks > ./env_setup\n" .
    "\t\@echo \"rm ./env_setup\" >> ./env_setup\n";
  if (!$nowarn) {
    $exeshellargs .= "\t\@echo\n" .
    "\t\@echo \"Please type . ./env_setup to set your environment\"\n" .
    "\t\@echo \"for memory debugging.  Run the debugger from the\"\n" .
    "\t\@echo \"shell you type this in.\"\n" .
    "\t\@echo\n" .
    "\t\@echo \"Note:  You _must_ have dmalloc properly set up in\"\n" .
    "\t\@echo \"       your shell environment for this library to\"\n" .
    "\t\@echo \"       function properly!\"\n" .
    "\t\@echo\n";
  }
}

# Now, write out the Makefile:
print MAKEFILE <<CONTENTS

DEBUG = $main::debug
PROFILE = $main::profile
BASE = $main::base
CC = $main::cc
CFLAGS = $main::cflags
EXEFILE = $exefile
INCLS = $main::incls
LIBS = $main::libs
VPATH = $vpath
DEFINES = $main::defines

OTHEROBJS = $otherobjs
MAINOBJS = $mainobjs
HEADERS = $headers

.c.o:
  \@\$(CC) -c \$(DEFINES) \$(CFLAGS) \$(INCLS) \$<
  \@echo "Compiling \$*.o"

\$(EXEFILE): \$(OTHEROBJS) \$(MAINOBJS) \$(HEADERS)
  \@echo "linking ..."
  \$(CC) \$(CFLAGS) \$(LIBS) -o \$(EXEFILE) \$(MAINOBJS) \$(OTHEROBJS)
CONTENTS
;
if ($exeshellargs) {
  print MAKEFILE "$exeshellargs\n";
```

```
}

print MAKEFILE <<CONTENTS

\$(MAINOBJS): \$(OTHEROBJS) $exesources
  \$(CC) -c \$(DEFINES) \$(CFLAGS) \$(INCLS) \$*.c

$miscdeps

clean:
  rm -f *.o core
CONTENTS
;

my(@work_list) = split(/:/, $vpath);
for (@work_list) {
  my(@work_list_2) = glob("$_/*.o");
  for (@work_list_2) {
    print MAKEFILE "\trm -f $_\n";
  }
}
print MAKEFILE "\n\n";

print "Done making makefile, cleaning old distribution and exiting.\n";
system("make clean > /dev/null");

exit(0);
```

graveyard

qdb_calculate.pl

```
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This program will be the overall master of all of the calculations that
# need to be done on the quantum database.  It will take the output from
# qdb_check (from stdin) and prepare and submit all of the jobs that
# need to be done from the output.  The general algorithm is as follows:
#  1)  Read the input and sort them into relevant arrays.  Read the
#      .qdb_checkrc file and process the directives.

# Note that the entire rest of the algorithm has been hashed and
# re-hashed.  The steps will be noted as we proceed through the program.

# Note that this program will be _very_ picky about what it gets from
# stdin, since it should be well defined as output from qdb_check.

# Included libraries
require('../general/rc_file_handling.pl');
use Fcntl ':flock';                  # import LOCK_* constants

# Global (frequently re_used) variables (here for reference).  Most
# variables are declared shortly before their first use.

$qdb_frag_count = 0;
$new_frag_count = 0;
@fragment = ();
@connectivity = ();
@torsion_list = ();
@frozen_dihedral_list = ();
@qcode_list = ();
$this_host = $ENV{"HOSTNAME"};

$line = <STDIN>;

$line =~
  /#There were (\d+) fragments in the database, and (\d+) new fragments/
  or die "Input to $0 is mangled, please use qdb_check to create " .
       "input for this program.\n";
$qdb_frag_count = $1;
$new_frag_count = $2;

for ($i = 0; $i < $qdb_frag_count; $i++) {
  # Do whatever we end up wanting to do with the qdb_fragments here
}

# Seek forward to the beginning of the new fragments
while (<STDIN> !~ /#Start new fragments/) { }

# The following block reads the structures and puts the information in
# multidimensional lists in the proper places.  It checks all Begin
# and End (as indicated by a "#") for mangling, making certain that the
# input is _exactly_ perfect.

$current_frag_number = 1;
while (<STDIN>) {

  $line = $_;
  #Seek forward to next fragment
  if ( $line !~ /#Fragment list molecule number $current_frag_number/) {
      next;
  }

  # Check the next line
  $line = <STDIN>;
  chomp($line);
  $line =~ /#/ && $line eq "#Begin coordinates" or die
    "Mangled input, please use qdb_check to generate input for $0\n";

  # Now, read the structure
  @work_list = ();
  while ( 1 ) {
    $line = <STDIN>;
    chomp($line);
    if ( $line =~ /#/ ) { last; }
    push( @work_list, $line );
  }
  $line eq "#End coordinates" or die
    "Mangled input, please use qdb_check to generate input for $0\n";

  # And push the molecule onto the fragment list
  push (@fragment, [ @work_list ]);

  # Do the same for the other three sets of data

  # Check the next line
  $line = <STDIN>;
  chomp($line);
  $line =~ /#/ && $line eq "#Begin connectivity" or die
  "Mangled input, please use qdb_check to generate input for $0\n";

  @work_list = ();
  while ( 1 ) {
    $line = <STDIN>;
    chomp($line);
    if ( $line =~ /#/ ) { last; }
    push( @work_list, $line );
  }
  $line eq "#End connectivity" or die
    "Mangled input, please use qdb_check to generate input for $0\n";
  # And push the molecule onto the connectivity list
  push (@connectivity, [ @work_list ]);

  # Check the next line
  $line = <STDIN>;
  chomp($line);
  $line =~ /#/ && $line eq "#Begin torsion_list" or die
    "Mangled input, please use qdb_check to generate input for $0\n";

  @work_list = ();
  while ( 1 ) {
    $line = <STDIN>;
    chomp($line);
    if ( $line =~ /#/ ) {last; }
    push( @work_list, $line );
  }
  $line eq "#End torsion_list" or die
    "Mangled input, please use qdb_check to generate input for $0\n";
  # And push the molecule onto the connectivity list
  push (@torsion_list, [ @work_list ]);

  # Check the next line
  $line = <STDIN>;
  chomp($line);
  $line =~ /#/ && $line eq "#Begin frozen_dihedral_list" or die
    "Mangled input, please use qdb_check to generate input for $0\n";

  @work_list = ();
  while ( 1 ) {
    $line = <STDIN>;
    chomp($line);
    if ( $line =~ /#/ ) { last; }
    push( @work_list, $line );
  }
  $line eq "#End frozen_dihedral_list" or die
    "Mangled input, please use qdb_check to generate input for $0\n";
  # And push the molecule onto the connectivity list
  push (@frozen_dihedral_list, [ @work_list ]);

  # Check the next line
  $line = <STDIN>;
  chomp($line);
  $line =~ /#/ && $line eq "#Begin qcode_list" or die
    "Mangled input, please use qdb_check to generate input for $0\n";

  @work_list = ();
  while ( 1 ) {
    $line = <STDIN>;
    chomp($line);
    if ( $line =~ /#/ ) {last; }
    push( @work_list, $line );
  }
  $line eq "#End qcode_list" or die
    "Mangled input, please use qdb_check to generate input for $0\n";
  # And push the molecule onto the connectivity list
  push (@qcode_list, [ @work_list ]);

  $current_frag_number++;
}

# Process the .qdb_checkrc file
open(RCFILE, '<.qdb_checkrc') or die
  "Unable to open .qdb_checkrc ... exiting\n";

$db_path = &read_scalar(RCFILE, "db_path");
defined($db_path) or die
```

```
  "Unable to find db_path in .qdb_checkrc file ... exiting\n";

$ab_initio_program = &read_scalar(RCFILE, "local_ab_initio_program");
defined($ab_initio_program) or die
  "Unable to find ab_initio_program in .qdb_checkrc file ... exiting\n";

$ab_initio_suffix = &read_scalar(RCFILE, "ab_initio_suffix");
defined($ab_initio_suffix) or die
  "Unable to find host_connect_method in .qdb_checkrc file ... exiting\n";

@hosts = &read_list(RCFILE, "available_hosts");
defined(@hosts) or die
  "Unable to find hosts in .qdb_checkrc file ... exiting\n";

$host_connect_method = &read_scalar(RCFILE, "host_connect_method");
defined($host_connect_method) or die
  "Unable to find host_connect_method in .qdb_checkrc file ... exiting\n";

$this_user = &read_scalar(RCFILE, "user_name");
defined($host_connect_method) or die
  "Unable to find this_user in .qdb_checkrc file ... exiting\n";

close(RCFILE);

# Ok, we are now at the point where all of the fragments are recorded, and
# all of the (for now) important variables from the .qdb_checkrc file are
# recorded.  We can go on with the next steps.

# The is_in_qdb() function from qdb_check guarantees that all of the
# fragments in frag_list are not in the db, so that will not be checked
# here.  (Note:  it's imperative that when that function is designed,
# it check the connectivity of any fragments with the same molecular
# formulae.  It must also output (with the fragments _in_ the qdb)
# which dihedrals need to be done, and which bonds need to be frozen.
# This script can then start only the torsions that are needed.

# Get unique directory base names for each fragment

@dir_name_list = ();
for ($i = 0; $i < $new_frag_count; $i++ ) {
  $dir_name_list[$i] = get_CHNO_formula( @{$fragment[$i]} );
}

# Determine if we've either crashed, or another instance of us is
# currently active (which means we have to look for duplicate
# fragments - slow!)
if ( -e "$db_path/control/qdb_crashed" ) { $recover = 1; }
else {
  # Create the file
  open (WORK_FILE, ">$db_path/control/qdb_crashed")
    or die "Could not create $db_path/control/qdb_crashed "
      .  "to save status ... exiting\n";
  close (WORK_FILE);
  $recover = 0;
}

# Get the contents of the db_path directory into a hash (will be much)
# faster to search as it grows
opendir(DB_DIR, $db_path) or die
  "Cannot open database directory \"$db_path\" ... exiting";

# Lock it (wait until we get lock)
# I learned (in a perl 5.6.0) that flock does not work for directories.
# in this case, we'll just have to hope that the directory isn't being
# written to while we're reading it.
# flock(DB_DIR, LOCK_EX);
@work_list = readdir(DB_DIR);

# Put the list in a hash
for (@work_list) { $db_directory{$_} = 1; }
closedir(DB_DIR);

# Remove the extraneous entries
delete ( $db_directory{"."} );
delete ( $db_directory{".."} );
delete ( $db_directory{"control"} );

# Now, make each base name unique
for ($i = 0; $i < $new_frag_count; $i++ ) {
  $do_loop = 1;
  for ( $j = 0; $do_loop; $j++ ) {
if ( $j < 0 ) {print "j = $j";}
    $this_dir_name = "$dir_name_list[$i]-$j";
    unless(exists($db_directory{$this_dir_name})) {
      $dir_name_list[$i] = $this_dir_name;

      # Write this directory immediately, since some fragments
      # may have the same base name.  Be certain to add it to the
      # hash %db_directory
      $db_directory{$this_dir_name} = 1;
      mkdir ("$db_path/$this_dir_name", 0770);

      $do_loop = 0;
    } elsif ($recover) {
      $no_file = 0;
      open(WORK_FILE, "<$db_path/$this_dir_name/Original_structure.raw")
        or $no_file = 1;
      open(WORK_FILE2, "<$db_path/$this_dir_name/Connectivity.raw")
        or $no_file = 1;
      if ($no_file) {
        # This directory is either hosed, or was produced earlier
        # in this loop, move on to the next one.
        # Note that we will need something to clean these
        # 'hosed' directories later
        next;
      }
      # Compare this fragment with the Original_structure.raw file
      @work_list = <WORK_FILE>;
      @work_list2 = <WORK_FILE2>;
      for (@work_list) {chomp($_);}
      for (@work_list2) {chomp($_);}

      if ( join("",@{$fragment[$i]}) eq join("",@work_list) &&
         join("",@{$connectivity[$i]}) eq join("",@work_list2) ) {
        $dir_name_list[$i] = $this_dir_name;
        last;
      } else {
        # Do nothing
      }
      close(WORK_FILE);
      close(WORK_FILE2);
    }
  }
}

# The usage of flock() _should_ have kept this directory locked, so long
# as any other program trying to use it also uses flock()
# See note on flock at the flock(DB_DIR, LOCK_EX) call (now commented out)
# flock(DB_DIR, LOCK_UN);

# Make the old directory names the full path names
for (@dir_name_list) { $_ = "$db_path/$_"; }

# For some reason, I couldn't get the permissions to become what they
# are set for.  I'm certain it has something to do with umask, though
# I can't say I know what it is.  Regardless, reset the permissions
# in an extra step for now.
for (@dir_name_list) { chmod (0770, $_); }

# Within each directory we just created, we need to create an:
# "Original_structure.raw" file, which contains _exactly_ what's in
# the fragment array for that fragment.  Likewise, we need to create
# a "Connectivity.raw" file that holds the connectivity.  Then, when
# first checking file names, each directory that has the same base
# name will be checked to see if the Original_structure.raw and
# "Connectivity.raw" files have identical information.  If they do,
# That directory name will be set to the corresponding number, and in
# the subsequent step, the .raw files will _not_ be generated.  This
# construction allows the master script to restart in case of a crash,
# so long as it gets _exactly_ the same input (it compares the geometry
# and order of the atoms, so it needs the same input).  This habit of
# 'leaving a trail of files' will be used with all of the associated
# programs.

# Initially condition the new directories
for ( $i = 0; $i < $new_frag_count; $i ++ ) {
  # The previous section that looked for uniqe directory names
  # may have given us a directory that was already conditioned.
  # Check this first
  if ( -e "$dir_name_list[$i]/Original_structure.raw" ) {
    next;
  }

  # Write original structure
  open(WORK_FILE, ">$dir_name_list[$i]/Original_structure.raw")
    or die "Could not open Original_structure.raw file for writing in " .
    "beginning conditioning new fragment directories ... exiting";

  print WORK_FILE join("\n", @{ $fragment[$i]} );
  print WORK_FILE "\n";

  close(WORK_FILE);

  # Write connectivity
  open(WORK_FILE, ">$dir_name_list[$i]/Connectivity.raw")
    or die "Could not open Original_structure.raw file for writing in " .
    "beginning conditioning new fragment directories ... exiting";

  print WORK_FILE join("\n", @{ $connectivity[$i]} );
  print WORK_FILE "\n";

  close(WORK_FILE);

  # Write qcodes
  open(WORK_FILE, ">$dir_name_list[$i]/Qcodes")
    or die "Could not open Qcodes file for writing in " .
    "beginning conditioning new fragment directories ... exiting";

  print WORK_FILE join("\n", @{ $qcode_list[$i]} );
  print WORK_FILE "\n";

  close(WORK_FILE);
}

# As of now, with an 18 frament input, the program takes 2 seconds to
# execute.  As this time increases, it will be important to provide the
# user with feedback as to what's happening.

# The first step is to prepare all of the jobs that need to be done, and
# to put the (local) input files in the proper directory.  This is done
# without regard as to what has happened before, since this program will
# not know the proper input name for the file, and the records may
# be very messed up from a previous crash.  Later, seeing if the job
# is done (with a callout to local_is_job_finished() ) will determine
# the actual state of the job.

for ( $i = 0; $i < $new_frag_count; $i ++ ) {
  $command_string = "format_for_${ab_initio_program}.pl ".
    "$dir_name_list[$i] Original_structure.raw Initial_optimization";
  system($command_string);
}

# All files are in the respective directories, now add them to the
# que --- The organization for this section has changed.  This program
# will submit all of the jobs to the control file, (with the current pid
# included), and the qdb_local_submit.pl will actually do the work.  For
# 'cleanlyness', this file won't put in any jobs that are already in the
# que section, but this isn't a problem, since the qdb_local_submit.pl
# program will not re-submit anything that has already been finished
# once.

$is_open = 1;
open(QUE, "<$db_path/control/que") or $is_open = 0;

# Before locking and submitting, we need to get a list of filenames, and
# see to it that we are not submitting duplicate jobs
undef(%que);
if ($is_open) {
```

```
  @work_list = <QUE>;
  close(QUE);
  for(@work_list) { chomp($_); }
  # Since only the 3rd item in the que is important, we will make another
  # list with only those
  for (@work_list) {
    ( $discard, $discard, $discard, $_ ) = split(",", $_);
  }

  # Put the list in a hash
  for (@work_list) { $que{$_} = 1; }
} else {
  # Make an empty que
  $que{""} = 0;
}

# The hash now necessarily has no repeats, and can be used to prevent
# adding duplicates

open(QUE, ">>$db_path/control/que") or die
  "Could not open que file in control directory of $db_path ... exiting\n";
flock(QUE, LOCK_EX);
seek(QUE, 0, 2);
for ( $i = 0; $i < $new_frag_count; $i ++ ) {
  $work_string = "$dir_name_list[$i]/Initial_optimization." .
           "$ab_initio_suffix";
  unless(defined($que{$work_string})) {
    print QUE "$this_user,$this_host,$$,$work_string\n";
  }
}
flock(QUE, LOCK_UN);

# Note something peculiar here.  The previous section keeps this program
# from submitting duplicate job requests, but in doing so, it fails to
# announce that it will be waiting for the jobs to finish.  This currently
# appears to be unavoidable.  As a result, the active part of the loop
# will not wait indefinately to be awakened, but will sleep for a moderately
# long period (10-30 minutes) between waking up and checking to see if
# its jobs are finished.  It will do this _only_ by checking for the
# expected log file for each unfinished job.  At that point it will make
# it's next decision.  Right after the sleep statement, check for the
# existence of "$db_path/control/message.<parent_pid>", and use that if
# it exists.  If it does not, do the searching manually.

# The following exit keeps the cleanup from happening, making the
# program 'beleive' it has crashed.
exit;
# Do cleanup here
unlink($db_path . "/control/qdb_crashed");

print "Program finished, exiting\n";
exit 0;

while (($TheKey, $TheVal) = each(%db_directory)) {
  print "$TheKey is the key for $TheVal\n";
}

print join("\n", @dir_name_list);
print "\n";

########################################################################
#       End of program                                                 #
########################################################################

# The following function looks through the array passed to it and
# the CwHxNyOz string for it's molecular formula.  If there are none
# of those atoms in the molecular formula, it returns "Other"
sub get_CHNO_formula {
  my(@molecule) = @_;
  my($C) = 0;
  my($H) = 0;
  my($N) = 0;
  my($O) = 0;
  my($label) = "";

  foreach $line (@molecule) {
    ( $label ) = split( /,/, $line, 2);
    # split returns a list ( the parentheses () ), so get the
    # uppercase of the element.  I have no idea why splitting into
    # 1 doesn't work, but 2 does ??
    $label = uc($label);

    if ( $label eq "C" ) { $C++; }
    elsif ( $label eq "H" ) { $H++; }
    elsif ( $label eq "N" ) { $N++; }
    elsif ( $label eq "O" ) { $O++; }
  }
  $label = "";
  if( $C != 0 ) {
    $label .= "C";
    if ($C >= 2 ) { $label .= $C; }
  }
  if( $H != 0 ) {
    $label .= "H";
    if ($H >= 2 ) { $label .= $H; }
  }
  if( $N != 0 ) {
    $label .= "N";
    if ($N >= 2 ) { $label .= $N; }
  }
  if( $O != 0 ) {
    $label .= "O";
    if ($O >= 2 ) { $label .= $O; }
  }
  if ( $label eq "" ) { return "Other"; }
  return $label;
}

# print "Hostlist:\n";
# print join("\n", @hosts);
# print "\n";

# The following demonstrates how to get a molecule out
# print "\n" . join("\n", @ {$fragment[0]} ) . "\n";
# print "\n" . join("\n", @ {$connectivity[0]} ) . "\n";
# print "\n" . join("\n", @ {$torsion_list[0]} ) . "\n";
# print "\n" . join("\n", @ {$frozen_dihedral_list[0]} ) . "\n";

# The following section is a (very simple) demonstration of basic signal
# handling.  The subtlties (which I don't really understand) come about
# When trying to deal with what was happening when the program was
# interrupted.  This should make it at least reasonably easy to run the
# program without worrying about the hanging up bit.

# use sigtrap 'handler', \&wake_up, ALRM;
# use sigtrap 'handler', \&handle_hup, HUP;
# $out = 0;
# while ( !($out) ) {
#   print "Sleeping for 30\n";
#   sleep(30);
# }

# The following is my attempt to play with signals
# sub wake_up {
#   print "Are you trying to wake me up?\n";
#   return;
# }

# The following is my attempt to play with signals
# sub handle_hup {
#   print "This is the hang_up handler\n";
#   $out = 1;
#   return;
# }
```

log2str

get_bond_order.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* This function takes as arguments 2 integers (atomic numbers of the       */
/* atoms in question, and the distance (in angstroms) between them.  It     */
/* returns the bond order based on atomic radii, as outlined at:            */
/* http://www.iumsc.indiana.edu/radii.html - which references:              */
/* N W Alcock, Bonding and Structure (Ellis Horwood, 1990).                 */
/* Finally, if the atoms are not bonded, or an insane distance (smaller than */
/* seems 'reasonable'), it returns -1.                                      */

#include <stdio.h>

/* Prototypes */
float get_cal_rad(float, float);
float myabs(float);
float mymin(float, float, float);

float get_bond_order (int atom1, int atom2, float distance) {

  float single_bond_radius[106]; /* Tables only go up to Iodine */
  float one_and_half_bond_radius[9];
  float double_bond_radius[9];
  float triple_bond_radius[8];
  float tolerance = 0.030; /* Tolerance is a % of the total bond (ideal) */
  float single_bond_distance, one_and_half_bond_distance,
    double_bond_distance, triple_bond_distance;
  float x, y, z;
  int i;

  if (atom1 > 105) {
    fprintf(stderr, "\nBad atom #%d passed to get_bond_order\n", atom1);
    return -1.0;
  }
  if (atom2 > 105) {
    fprintf(stderr, "\nBad atom #%d passed to get_bond_order\n", atom2);
    return -1.0;
  }
  /* Initializations of 'hard' data */
  for (i = 0; i < 106; i++) {single_bond_radius[i] = 0;}
  single_bond_radius[1] = 0.299;
  /* Note:  It has been found that C-H bonds seem to be significantly */
  /* longer than other X-H bonds, tweak the radius if we're talking */
  /* about a C-H bond */
  if ( (atom1 == 1 && atom2 == 6) || (atom2 == 1 && atom1 == 6) ) {
    single_bond_radius[1] = 0.330;
  }
```

```
  single_bond_radius[4] = 1.06;  /* This value is suspect */
  single_bond_radius[5] = 0.83;
  single_bond_radius[6] = 0.767;
  single_bond_radius[7] = 0.702;
  single_bond_radius[8] = 0.626;  /* The old value of 0.659 has been */
                /* adjusted to deal with the many */
                /* O-C-F combinations in which the */
                /* C-O is significantly shorter.  The */
                /* range of values from a series of */
                /* polyfluorinated dimethyl ethers */
                /* (Calc'd from B3LYP opt) was used to */
                /* calibrate this the maximum variance */
                /* in the set was +- 2.66% */

  /* Note:  Oxygen bonds can be a fair bit longer to carbons on rings */
  /* if one of the atoms is an oxygen, we will increase the tolerance */
  /* to catch these cases */
  if (atom1 == 8 || atom2 == 8) { tolerance = 0.045; }

  single_bond_radius[9] = 0.595;  /* Note: the old value of 0.619; is */
                /* valid _only_ or monofluorides. The */
                /* new value is better centered for */
                /* between mono and trifluorides.  If */
                /* this _still_ doesn't work, setting */
                /* the tolerance (when fluorine is */
                /* present) to 5% should definately */
                /* take care of the problem */
  single_bond_radius[13] = 1.18;
  single_bond_radius[14] = 1.090;
  single_bond_radius[15] = 1.088;
  single_bond_radius[16] = 1.052;
  single_bond_radius[17] = 1.023;
  single_bond_radius[31] = 1.25;
  single_bond_radius[32] = 1.22;
  single_bond_radius[33] = 1.196;
  single_bond_radius[34] = 1.203;
  single_bond_radius[35] = 1.199;
  single_bond_radius[49] = 1.41;
  single_bond_radius[50] = 1.39;
  single_bond_radius[51] = 1.37;
  single_bond_radius[52] = 1.391;
  single_bond_radius[53] = 1.395;

  /* Initialize multiples bond radii for C, N, and O */
  one_and_half_bond_radius[6] = 0.692; /* From RHF optimization of benzene */
  one_and_half_bond_radius[7] = 0.629; /* From RHF optimization of pyridine */
  one_and_half_bond_radius[8] = 0.561; /* From RHF optimization of nitrobenzene
*/
  double_bond_radius[6] = 0.661;
  double_bond_radius[7] = 0.618;
  double_bond_radius[8] = 0.549;
  triple_bond_radius[6] = 0.591;
  triple_bond_radius[7] = 0.545;

  /* Verify that atom #'s are supported. */
  if (single_bond_radius[atom1] == 0) {
    fprintf(stderr, "\nBad atom #%d passed to get_bond_order\n", atom1);
    return -1.0;
  }
  if (single_bond_radius[atom2] == 0) {
    fprintf(stderr, "\nBad atom #%d passed to get_bond_order\n", atom2);
    return -1.0;
  }

  /* First, construct the 'ideal' bond distance */
  single_bond_distance = single_bond_radius[atom1] +
    single_bond_radius[atom2];

  /* If the distance is greater than the single bond tolerance (the */
  /* most common case), return -1 immediately */
  if (distance > (single_bond_distance * (1 + tolerance))) {return -1.0;}

  /* If the function hasn't returned, and the distance is greater than */
  /* the minimum tolerance, return 1 */
  if (distance >= single_bond_distance * (1 - tolerance)) {return 1.0;}

  /* If either of the atoms are monovalent, and 1 hasn't been returned, */
  /* return no bond */
  if (atom1 == 1 || atom1 == 9 || atom1 == 17 || atom1 == 35 ||
    atom1 == 53 || atom2 == 1 || atom2 == 9 || atom2 == 17 ||
    atom2 == 35 || atom2 == 53) {return -1.0;}

  /* Note that if there were a single bond, that value has been */
  /* returned.  For the other cases, we will not use a 'simple' */
  /* cutoff.  Instead, we will compare how close the actual value */
  /* is to calculated values, and return the best match */

  /* Now, check for multiple bond references, and set them (use */
  /* Default polynomial fit if there are not analytical numbers */
  if (atom1 == 6 || atom1 == 7 || atom1 == 8) {
    x = one_and_half_bond_radius[atom1];
  }
  else {x = get_cal_rad(single_bond_radius[atom1], 1.5);}
  if (atom2 == 6 || atom2 == 7 || atom2 == 8) {
    y = one_and_half_bond_radius[atom2];
  }
  else {y = get_cal_rad(single_bond_radius[atom2], 1.5);}
  one_and_half_bond_distance = x + y;

  /* Now get double bond distance */
  if (atom1 == 6 || atom1 == 7 || atom1 == 8) {
    x = double_bond_radius[atom1];
  }
  else {x = get_cal_rad(single_bond_radius[atom1], 2.0);}
  if (atom2 == 6 || atom2 == 7 || atom2 == 8) {
    y = double_bond_radius[atom2];
  }
  else {y = get_cal_rad(single_bond_radius[atom2], 2.0);}
  double_bond_distance = x + y;

  /* Finally, get triple bond distance, note that bond order 2.5 */
  /* makes no sense except in the context of hypervalence, which */
  /* should not be a concern for us */
  if (atom1 == 6 || atom1 == 7) {
    x = triple_bond_radius[atom1];
  }
  else {x = get_cal_rad(single_bond_radius[atom1], 3.0);}
  if (atom2 == 6 || atom2 == 7) {
    y = triple_bond_radius[atom2];
  }
  else {y = get_cal_rad(single_bond_radius[atom2], 3.0);}
  triple_bond_distance = x + y;

  /* If we _could_ be dealing with a multiple bond, continue */
  if (distance < (triple_bond_distance * (1 - tolerance))) {
    fprintf(stderr, "\nWarning:  Atoms %d-%d too close for"
    " triple bond in get_bond_order.\n", atom1, atom2);
    return -1.0;
  }

  /* And finally, find the bond type with minimum deviation */
  x = myabs(one_and_half_bond_distance - distance);
  y = myabs(double_bond_distance - distance);
  z = myabs(triple_bond_distance - distance);

  /* As much as it sux to have to add new rules, it's important to know */
  /* that the idea of a 1 1/2 bond for O is only sensible in the */
  /* context of a nitro group.  If there's an O and not an N, do the */
  /* following.  Also, generate no warnings in this case, O does strange */
  /* overlap that makes it frequently anomylous (carbonyl's) */
  if ( (atom1 == 8 && atom2 != 7) || (atom1 != 7 && atom2 == 8) ) {
    /* We need to return either a single or double bond ... no */
    /* other cases are possible */
    x = myabs(single_bond_distance - distance);
    if ( x < y ) { return 1.0; }
    else { return 2.0; }
  }

  /* But before we actually return the bond order, make certain */
  /* The minimum deviation is within our defined value - if not, */
  /* print a warning to stderr */
  if ( mymin(x, y, z) > tolerance) {
    /* It has become apparent that C-C bonds in aromatic rings */
    /* are somewhat troublesome.  Suppress this warning if it */
    /* would refer to a C-C 1 1/2 bond */
    if ( ! (atom1 == 6 && atom2 == 6 && (x == mymin(x, y, z) ) ) ) {
      fprintf(stderr,
        "Warning: assigning bond %d-%d outside of tolerance: %f\n"
        "deviations are oah-d-t %g - %g - %g\n",
        atom1, atom2, distance, x, y, z);
    }
  }

  if (x <= y && x <= z) return 1.5;
  if (y <= z) return 2.0;
  return 3.0;

}


/* The following function uses the 'magic' polynomial fit that was */
/* worked out by Josh Radke to return what a non-single bond should */
/* return.  It may need to be refined to produce more consitent results */

float get_cal_rad(float start_d, float bond_order) {

  float a, b, c; /* From the expression ax^2 + bx + c = 0 */

  a = 0.0362;
  b = -0.2588;
  c = 1.2111;

  return (start_d * (a * bond_order*bond_order + b * bond_order + c));

}

/* Was uncertain of where to find this function, so re-wrote quickly */
float myabs (float value) {
  return ( (value <= 0) ? (-1 * value) : value );
}

/* In an effort to avoid too much confusion, separated this small task */
/* into a function */
float mymin(float a, float b, float c) {

  float min;
  min = ( a < b ? a : b);
  min = ( min < c ? min : c);

  return min;
}
```

one_timers

repair_qdb_charges.pl

```
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.
```

```
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# IMPORTANT NOTE:  This was also flawed, as the --Link1-- correction
# failed to include a nosymm option, resulting in long and incorrect
# results, see repair_db-2.pl for the coreect solution

# This one timer is made to repair the qdb, when the .com files were
# incorrectly created.  In this case, the charges were calculated
# wrong, and each of the calculations must be re-done with the link1
# section reading:
# #p RB3LYP/6-311+g(2d,p) test pop=chelpg geom=allcheck guess=read
# instead of: #p RB3LYP/6-311+g(2d,p) test pop=chelpg geom=allcheck
guess=(only,read)
#
# Each of the *.com files will be re_written, and then they will be
# written to the que file in the database for processing by
# qdb_local_submit.pl.

BEGIN {
  # Since our own modules aren't properly installed, add to the INC
  # list at compile time
  push(@INC, "../perl_modules");
}

# Included libraries
require('../perl_modules/g98_functions.pl');
use NETFLOCK qw(:basic);

# pragmas
use strict;

# Function prototypes
sub get_all_comfilenames($);
sub recurse_get_coms($);

# Begin program

print "Building list of files to correct:\n";
my(@comfiles) = get_all_comfilenames("/private_ffd/qdb");

print "Updating com files:\n";

foreach (@comfiles) {

  my($thisfile) = $_;
  my(@new_com) = ();
  my($line);
  my($is_torsion) = 0;

  print "Preparing submission for $thisfile\n";

  # The first step is to read the geometry from the log file.
  my($basename, undef) = split('\.', $thisfile, 2);
  my($logname) = "$basename.log";

  my(@molecule) = get_optimized_structure($logname);

  my(@old_com);
  open(TMP, "<$thisfile") or die "Unable to open $thisfile";
  @old_com = <TMP>;
  close(TMP);

  until ( ($line = shift(@old_com) ) =~ /^0 1/ or
       $line =~ /^.*geom=modredundant/ ) {
  push(@new_com, $line);
  }

  # If we stopped early because it was a torsion type of calculation,
  # set our flag and fix the route line.
  if ($line =~ /^(.*)geom=modredundant/) {
  $is_torsion = 1;

  # And fast forward as if we were a normal file
  push(@new_com, $line);
  until (($line = shift(@old_com) ) =~ /^0 1/) {
    push(@new_com, $line);
  }
  }

  push(@new_com, $line);

  foreach (@molecule) {
  push(@new_com, join(",", @{$_}) . "\n");
  }
  push(@new_com, "\n");

  # Fast forward old file
  until (shift(@old_com) =~ /^[ \t]*$/) { }

  # If we're dealing with one of the torsions files, we need to modify
  # the particular entry that says +=, we'll simply freeze that bond.
  if ($is_torsion) {

  until (($line = shift(@old_com)) =~ /^[ \t]*$/) {
    if ( $line =~ /^(.*)+=.*F/) {
    $line =~ /^(.*)\+=.*F/;

    $line = $1 . "F\n";
    }
    push(@new_com, $line);
  }
  push(@new_com, "\n");
  }

  while ( $line = shift(@old_com) ) {
  if ($line =~ /^(.*)guess=\(only,read\)(.*)/) {
    $line="$1guess=read$2\n";
  }
  push(@new_com, $line)
  }

  # The new com file is finished, re_write it, delete the previous log
  # file, and submit it to the que.

  open(TMP, ">$thisfile") or die "Unable to open $thisfile";
  print TMP @new_com;
  close(TMP);

  unlink("$logname");

  nflock("/private_ffd/qdb/control/que");
  open(QUE, ">>/private_ffd/qdb/control/que") or
  die "Unable to open /private_ffd/qdb/control/que for appending";
  print QUE "radke,jubjub,$$,$thisfile\n";
  close QUE;
  nfunlock("/private_ffd/qdb/control/que");

}

print "Successfully updated " . scalar(@comfiles) . " files\n";

exit 0;


######################################################################
#       End of program                                               #
######################################################################

# Functions

# This function retrieves (recursively) all filenames that end with
# .com, and have a corresponding .log file.
sub get_all_comfilenames($) {

  my($basepath) = shift;

  my(@comlist);

  @comlist = recurse_get_coms($basepath);

  return @comlist;

}

sub recurse_get_coms($) {

  my($this_dir) = shift;

  chdir("$this_dir");

  opendir(DIR, ".");

  my(@full_dir_list) = readdir(DIR);
  my(@dirs_to_visit, @coms_to_add);

  foreach (@full_dir_list) {

  if ($_ eq '.' or $_ eq '..') {
    next;
  }

  if ( -d $_ and -r $_ and -w $_ and -x $_ ) {
    push(@dirs_to_visit, "$_");
    next;
  }

  if ( $_ =~ /(.*)\.com$/ and -e "$1.log") {
    push(@coms_to_add, "$this_dir/$_");
    next;
  }
  }

  foreach (@dirs_to_visit) {
  push(@coms_to_add, recurse_get_coms("$this_dir/$_"));
  }

  return (@coms_to_add);


}
```

repair_db-2.pl

```
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
```

```
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This one timer is made to repair the qdb, when the .com files were
# incorrectly created.  In this case, the charges were calculated
# wrong, and each of the calculations must be re-done with the link1
# section reading:
# #p RB3LYP/6-311+g(2d,p) test pop=chelpg geom=allcheck guess=read
# instead of: #p RB3LYP/6-311+g(2d,p) test pop=chelpg geom=allcheck
guess=(only,read)
#
# Each of the *.com files will be re_written, and then they will be
# written to the que file in the database for processing by
# qdb_local_submit.pl.

BEGIN {
  # Since our own modules aren't properly installed, add to the INC
  # list at compile time
  push(@INC, "../perl_modules");
}

# Included libraries
require('../perl_modules/g98_functions.pl');
use NETFLOCK qw(:basic);

# pragmas
use strict;

# Function prototypes
sub get_all_comfilenames($);
sub recurse_get_coms($);

# Begin program

print "Building list of files to correct:\n";
my(@comfiles) = get_all_comfilenames("/private_ffd/qdb");

print "Updating com files:\n";

foreach (@comfiles) {

  my($thisfile) = $_;
  my(@new_com) = ();
  my($line);
  my($is_torsion) = 0;

  print "Preparing submission for $thisfile\n";

  # The first step is to read the geometry from the log file.
  my($basename, undef) = split('\.', $thisfile, 2);
  my($logname) = "$basename.log";

  unless (-e $logname) {
  print STDERR "No logfile for $basename.com exists!  It may be " .
    "necessary to restore the database and re-run this program\n";
  }

  my(@molecule) = get_optimized_structure($logname);

  my(@old_com);
  open(TMP, "<$thisfile") or die "Unable to open $thisfile";
  @old_com = <TMP>;
  close(TMP);

  until ( ($line = shift(@old_com) ) =~ /^0 1/ or
       $line =~ /^.*geom=modredundant/ ) {
  push(@new_com, $line);
  }

  # If we stopped early because it was a torsion type of calculation,
  # set our flag and fix the route line.
  if ($line =~ /^(.*)geom=modredundant/) {
  $is_torsion = 1;

  # And fast forward as if we were a normal file
  push(@new_com, $line);
  until (($line = shift(@old_com) ) =~ /^0 1/) {
    push(@new_com, $line);
  }
  }

  push(@new_com, $line);

  foreach (@molecule) {
  push(@new_com, join(",", @{$_}) . "\n");
  }
  push(@new_com, "\n");

  # Fast forward old file
  until (shift(@old_com) =~ /^[ \t]*$/) { }

  # If we're dealing with one of the torsions files, we need to modify
  # the particular entry that says +=, we'll simply freeze that bond.
  if ($is_torsion) {

  until (($line = shift(@old_com)) =~ /^[ \t]*$/) {
    if ( $line =~ /^(.*)+=.*F/) {
    $line =~ /^(.*)\+=.*F/;

    $line = $1 . "F\n";
    }
    push(@new_com, $line);
  }
  push(@new_com, "\n");
  }

  while ( $line = shift(@old_com) ) {
  if ($line =~ /^(.*)guess=read(.*)/) {
    $line="$1guess=read nosymm$2\n";
  }
  push(@new_com, $line)
  }

  # The new com file is finished, re_write it, delete the previous log
  # file, and submit it to the que.

  open(TMP, ">$thisfile") or die "Unable to open $thisfile";
  print TMP @new_com;
  close(TMP);

  unlink("$logname");

  nflock("/private_ffd/qdb/control/que");
  open(QUE, ">>/private_ffd/qdb/control/que") or
  die "Unable to open /private_ffd/qdb/control/que for appending";
  print QUE "radke,jubjub,$$,$thisfile\n";
  close QUE;
  nfunlock("/private_ffd/qdb/control/que");

}

print "Successfully updated " . scalar(@comfiles) . " files\n";

exit 0;


######################################################################
#       End of program                                               #
######################################################################

# Functions

# This function retrieves (recursively) all filenames that end with
# .com.
sub get_all_comfilenames($) {

  my($basepath) = shift;

  my(@comlist);

  @comlist = recurse_get_coms($basepath);

  return @comlist;

}

sub recurse_get_coms($) {

  my($this_dir) = shift;

  chdir("$this_dir");

  opendir(DIR, ".");

  my(@full_dir_list) = readdir(DIR);
  my(@dirs_to_visit, @coms_to_add);

  foreach (@full_dir_list) {

  if ($_ eq '.' or $_ eq '..') {
    next;
  }

  if ( -d $_ and -r $_ and -w $_ and -x $_ ) {
    push(@dirs_to_visit, "$_");
    next;
  }

  if ( $_ =~ /(.*)\.com$/ and -e "$1.log") {
    push(@coms_to_add, "$this_dir/$_");
    next;
  }
  }

  foreach (@dirs_to_visit) {
  push(@coms_to_add, recurse_get_coms("$this_dir/$_"));
  }

  return (@coms_to_add);

}
```

perl_modules

g98_functions.pl

```
# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

```
# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This is a library for handling functions specific to g98 (parsing and
# such).  Other ab_initio packages may use a slightly different set of
# functions, and should be easy to add to another library.

# Note that this library also uses LINALG.pm.  Since this library
# can be 'use'ed from anywhere, it's the responsibility of the program
# that includes it to require this as well.  (The calling program will
# need to add the LINALG directory to @INC

# Note also that this library uses ../general/rc_file_handling.pl, and
# the program that uses it must also load that library

# Original work on this file was finished eons ago (late 2000).  The
# following functions are added to support torsion_driver.pl in 8/2001

# Prototypes.  Since this is an old style library, it cannot be 'use' ed
# like a module.  Further, the functions must be prototyped in the
# program that 'require' s this module, these are here for references, and
# can simply be copied and pasted.
sub get_last_dihedral_and_energy($\@);
sub get_optimized_structure($);
sub find_end_of_optimization (*);
sub atomic_number_to_label($);
sub atomic_label_to_number($);
sub hartree_to_kcal_per_mole($);
sub make_dihed_inp_file($$\@\@\@$);
sub read_first_geometry($);
sub extract_chelpg_charges($);

# sub is_finished() is old style, and I won't enforce prototypes, as other
# programs are currently using it.

# The following function digs the requested information out of the specified
# .log file.  It returns a string of the format "<angle> <energy>".  If there
# are any errors, it returns undef.
sub get_last_dihedral_and_energy($\@) {
  my($filename) = shift;
  my($dihedref) = shift;
  my($i);
  my(@dihedral) = @$dihedref;
  my($line);
  my(@molecule);

  # Make dihedral base one, like g98 uses.
  @dihedral = map { $_ + 1 } @dihedral;

  # Next, we need to get the coordinates of each of the atoms in the
  # dihedral.

  open(TMP, "<$filename") or return undef;
  (my($curpos) = find_end_of_optimization(TMP)) or return undef;

  my($found_start) = 0;
  while ( $line = <TMP> ) {
  if ( $line =~
     /^
       \s*(\d+)      # Atomic center number
       \s+(\d+)      # Atomic number
       \s+\d+\s+      # Atomic type?
       (-?\d+\.\d*)\s+    # X Coordinate
       (-?\d+\.\d*)\s+    # Y Coordinate
       (-?\d+\.\d*)    # Z Coordinate
     $/x ) {
    $found_start = 1;
    # Save this line, and all of the rest we come into
    $molecule[$1][0] = atomic_number_to_label($2);
    $molecule[$1][1] = $3;
    $molecule[$1][2] = $4;
    $molecule[$1][3] = $5;
    # Note that unlike in a similar section later, we leave the
    # molecule one based;

  } else {
    if ($found_start) {
    last;
    }
  }
  }

  # If there's only one step in the optimization, or the structure
  # was 'too big;, there will be no structure following "--
  # Stationary point found".  If this is the case, we need to get
  # the geometry from somewhere else.  Frequently, we run jobs that
  # in the final step read the geometry from the checkpoint file,
  # we'll look for that geomtry in this case.
  unless ($molecule[1]) {

  seek(TMP, $curpos, 0);

  # Now look for another common geometry identifier.
  while ( $line = <TMP> ) {
    unless ( $line =~ /^\s*Redundant internal coordinates taken from/) {
    next;
    }

    # Eat 2 lines
    <TMP>; <TMP>;
    # Now read the geometry for as long as it lasts
    my $this_atom_index = 1;
    while ( $line = <TMP> ) {
    if ( $line =~
       /^
        \s*([A-Za-z]+),    # Label
        \d+,          # Discard this value
        (-?\d+\.\d*),    # X Coordinate
        (-?\d+\.\d*),    # Y Coordinate
        (-?\d+\.\d*)    # Z Coordinate
       $/x) {

      $molecule[$this_atom_index][0] = $1;
      $molecule[$this_atom_index][1] = $2;
      $molecule[$this_atom_index][2] = $3;
      $molecule[$this_atom_index][3] = $4;
      $this_atom_index++;
    } else {
      last;
    }
    }
  }
  }

  unless ($molecule[1]) {
  die "Unable to read geometry from $filename";
  }

  # We have the molecule, let's get the actual coordinates
  my(@dihed_coords);
  for $i ( 0 .. 3 ) {
  $dihed_coords[$i] = [ $molecule[$dihedral[$i]][1],
            $molecule[$dihedral[$i]][2], $molecule[$dihedral[$i]][3] ];
  }

  # We have the relvant coordinates, now get the last occurence of
  # SCF Done in the file

  # Only seek back a fair bit, we only need to get slightly before
  # here when we passed the last SCF done.  Note:  In the calculations
  # we've been doing, we do a single point calculation _after_ the
  # last optimization.  The reduction of position is only really
  # necessary when one takes the energy from the optimization, but
  # it's there in case someone changed the .qdb_checkrc and requests
  # no second single point energy.
  $curpos -= 50000;

  # Re-open the file in case it was closed on us by a call to
  # read_first_geometry().
  open(TMP, "<$filename") or return undef;

  seek(TMP, $curpos, 0);
  my($last_energy_line) = undef;

  while ( $line = <TMP> ) {
  if ( $line =~ /SCF Done:/ ) {
    $last_energy_line = $line;
  }
  }
  close(TMP);

  return undef unless(defined($last_energy_line));

  # And dig out the actual energy
  my(@work_list) = split(/ +/, $last_energy_line);
  while ( $work_list[0] !~ /^-?\d*\.\d*$/ ) {
  shift(@work_list);
  }
  my($energy) = $work_list[0];

  # Finally, change the units of energy to Kcal/mol
  $energy = hartree_to_kcal_per_mole($energy);

  # Now the tricky part, find the dihedral.  This is no longer tricky,
  # it is cleanly implemented in LINALG, my own module that must be
  # imported by whoever uses these functions.  I don't know how to
  # check what modules are active, (from within a module) but this
  # library should also have a begin section that checks to make sure
  # this module is in the namespace.

  my($dihedral_angle) = c_get_dihedral(@dihed_coords);

  return undef unless(defined($dihedral_angle));

  # And, return the string (in degrees).
  return $dihedral_angle * 180 / acos(-1)  . # acos(-1) = PI
  " " . $energy;

}

# This function does a simple conversion
sub hartree_to_kcal_per_mole($) {
  my($oldenergy) = shift;

  return $oldenergy * 627.5095;
}

# This function extracts the structure from a log file, it returns it
# as a list with the following structure:
# $retval[atom#][0]      <--- Atom's label
# $retval[atom#][1]      <--- Atom's x coordinate
# $retval[atom#][2]      <--- Atom's y coordinate
# $retval[atom#][3]      <--- Atom's z coordinate
sub get_optimized_structure($) {

  my($file) = shift;
  my($line);
  my(@molecule) = (undef);
  my($do_loop) = 1;

  open(TMP, "<$file") or
  die "Unable to open $file for reading";

  (my($curpos) = find_end_of_optimization(TMP)) or
  die "Unable to find end of optimization section of " .
    "$file in get_optimized_structure()";

  my($found_start) = 0;
  while ( $line = <TMP> ) {
  if ( $line =~
     /^
       \s*(\d+)      # Atomic center number
       \s+(\d+)      # Atomic number
       \s+\d+\s+      # Atomic type?
       (-?\d+\.\d*)\s+    # X Coordinate
       (-?\d+\.\d*)\s+    # Y Coordinate
       (-?\d+\.\d*)    # Z Coordinate
     $/x ) {
```

```
    $found_start = 1;
    # Save this line, and all of the rest we come into
    $molecule[$1 - 1][0] = atomic_number_to_label($2);
    $molecule[$1 - 1][1] = $3;
    $molecule[$1 - 1][2] = $4;
    $molecule[$1 - 1][3] = $5;
  } else {
    if ($found_start) {
    last;
    }
  }

  unless ($molecule[1]) {

    seek(TMP, $curpos, 0);

    # Now look for another common geometry identifier.
    while ( $line = <TMP> ) {
    unless ( $line =~ /^\s*Redundant internal coordinates taken from/) {
      next;
    }

    # Eat 2 lines
    <TMP>; <TMP>;
    # Now read the geometry for as long as it lasts
    my $this_atom_index = 0;
    while ( $line = <TMP> ) {
      if ( $line =~
          /^
        \s*([A-Za-z]+),    # Label
        \d+,          # Discard this value
        (-?\d+\.\d*),    # X Coordinate
        (-?\d+\.\d*),    # Y Coordinate
        (-?\d+\.\d*)    # Z Coordinate
       $/x) {

      $molecule[$this_atom_index][0] = $1;
      $molecule[$this_atom_index][1] = $2;
      $molecule[$this_atom_index][2] = $3;
      $molecule[$this_atom_index][3] = $4;
      $this_atom_index++;

      } else {
      last;
      }
    }
    }
  }

  unless ($molecule[1]) {
  die "Unable to read geometry from $filename";
  }


  }

  close(TMP);

  return (@molecule);
}

# The following function is the 'climax' of the work that the
# torsion_driver.pl function does.  A more useful prototype would be:
# make_dihed_inp_file(<($)new_file_base_name>,<($)new_angle>,
#                   <(\@)dihedral_of_interest>, <(\@)fragment geometry>,
#                   <(\@)frozen_bonds>,<$rcfilename>)
# It generates a gaussian .com input file in the current directory, writes
# it, and returns it's name
sub make_dihed_inp_file($$\@\@\@$) {
  my($input_file) = shift(@_) . ".com";
  my($angle) = shift;
  my(@mod_dihed) = @{$_[0]}; shift;
  my(@molecule) = @{$_[0]}; shift;

  my(@frozen_bonds) = @{$_[0]}; shift;
  my($config_file) = shift;

  # Other function wide variables
  my($i);

  # First, let's change all of the specifications to 1 based, since
  # that's what is native to gaussian
  foreach ( @mod_dihed ) {
  $_++;
  }
  ;

  for ( $i = $#molecule + 1; $i >= 1; $i-- ) {
  $molecule[$i] = $molecule[$i - 1];
  }
  $molecule[0] = undef;

  foreach (@frozen_bonds) {
  foreach (@{$_}) {
    $_++;
  }              # The first $_ is for the outside loop,
  }                           # the second increments the inside value

  # Ok, we're one based.  Find the dihedral angle (old)
  my(@qualified_dihedral);
  for $i ( 0 .. 3 ) {
  $qualified_dihedral[$i] = [ $molecule[$mod_dihed[$i]][1],
              $molecule[$mod_dihed[$i]][2],
              $molecule[$mod_dihed[$i]][3]  ];
  }

  my($old_dihedral) = c_get_dihedral(@qualified_dihedral)  * 180 / acos(-1);

  # And, get our dihedral shift
  my($dihed_shift) = $angle - $old_dihedral;
  if ( $dihed_shift < 0 ) {
  $dihed_shift += 360;
  }
  ;
```

```
  # Finally, since we can (and should, since it's easier to read, and
  # less error_prone) use wildcards for dihedral input, we will reduce
  # the @frozen_bonds list to the bond centered notations
  my(%frozen_dihed) = ();
  foreach (@frozen_bonds) {
  my(undef, $a1, $a2, undef) = @{$_};
  $frozen_dihed{"$a1 $a2"} = 1;
  }

  # Initialize values from .qdb_checkrc
  open(RCFILE, "<$config_file") or die
  "Unable to open $config_file in make_dihed_inp_file()\n";

  my($preopt_method) = &read_scalar(RCFILE, "preopt_method");
  defined($preopt_method) or die
  "Unable to find preopt_method in .qdb_checkrc file ... exiting\n";

  my($preopt_basis) = &read_scalar(RCFILE, "preopt_basis");
  defined($preopt_basis) or die
  "Unable to find preopt_basis in .qdb_checkrc file ... exiting\n";

  my($final_method) = &read_scalar(RCFILE, "final_method");
  defined($final_method) or die
  "Unable to find final_method in .qdb_checkrc file ... exiting\n";

  my($final_basis) = &read_scalar(RCFILE, "torsion_final_basis");
  defined($final_basis) or die
  "Unable to find final_basis in .qdb_checkrc file ... exiting\n";

  my($special_flags) = &read_scalar(RCFILE, "special_flags");
  defined($special_flags) or die
  "Unable to find special_flags in .qdb_checkrc file ... exiting\n";

  close(RCFILE);

  # Finally, we can prepare the file, but before we do, we need to
  # launder our input file.
  ($input_file) = $input_file =~ m%^([-./\\\w]+)%;

  open(OUTFILE, ">$input_file") or
  die "Unable to open $input_file for writing";

  print OUTFILE <<GFILE;
%mem=128MB
%chk=master.chk

#p $final_method/$final_basis//$preopt_method/$preopt_basis test $special_flags
geom=modredundant

No comment

0 1
GFILE
  # Now output the connectivity
  for $i ( 1 .. $#molecule ) {
  print OUTFILE join(",", @{$molecule[$i]}) . "\n";
  }

  # In gaussian's modredundant options, we'll be using the += syntax
  # for the central dihedral, so we need to know what the old_dihedral
  # was.

  # Finally, add the modredundant information
  print OUTFILE "\n";
  foreach (keys(%frozen_dihed)) {
  print OUTFILE "* $_ * F\n";
  }

  # And our own special dihedral
  print OUTFILE "* $mod_dihed[1] $mod_dihed[2] * A\n";
  print OUTFILE "* $mod_dihed[1] $mod_dihed[2] * += ";
  printf OUTFILE "%.5f", $dihed_shift;
  print OUTFILE " F\n\n";

  print OUTFILE <<GFILE;
--Link1--
%mem=128MB
%chk=master.chk
%nosave

#p $final_method/$final_basis $special_flags scf=tight pop=chelpg guess=read
geom=allcheck

GFILE

  close (OUTFILE);

  return $input_file;

}

# This function extracts the optimized connectivity from a log file as a
# hash with the following structure:
# The keys are the actual atom numbers (translated to 0 based), and the
# values are references to lists of the connected atoms.
sub get_optimized_connectivity($) {

  my($file) = shift;
  my($line);
  my(%connectivity);
  my($do_loop) = 1;
  my($found_start) = 0;

  open(TMP, "<$file") or
  die "Unable to open $file for reading";

  find_end_of_optimization(TMP) or
  die "Unable to find end of optimization section of " .
    "$file in get_optimized_structure()";

  while ( <TMP> !~ /^\s*!\s+Name\s+Definition\s+Value/ ) {
  }
  # Throw away the next line
```

```
  <TMP>;
  while ( <TMP> =~ /^\s*!\s*R\d+\s+R\((\d+),(\d+)/ ) {
  push ( @{$connectivity{$1 - 1}}, ($2 - 1));
  push ( @{$connectivity{$2 - 1}}, ($1 - 1));
  }

  close(TMP);

  # Note:  This is not guaranteed to return a 'perfect' connectivity.
  # gaussian may add redundant internal coordinates at will, or there
  # may (but hopefully isn't!) be dummy atoms.  Regardless, we'll use
  # it with it's current limitations.  If we wanted to get perfect
  # connectivity, we would be best off doing it ourselves (by making
  # XSUB's to some of the atom_handling.c routines)

  # In several tests, it works perfectly fine.

  return (%connectivity);
}
# This function takes an open filehandle, rewinds it to the beginning,
# and seeks forward until it finds the section of a gaussian .log file
# that look like this:
# -- Stationary point found
# It returns the file position indicator if the section is found, and
# undef if it's not

sub find_end_of_optimization (*) {

  my($logfile) = shift;
  my($i);
  my($line);

  seek($logfile, 0, 0);

 READFILE: while ( $line  = <$logfile> ) {
  if ( $line =~ /^\s+-- Stationary point found\.$/ ) {
    # We found it, return a true value
    return tell $logfile;
  }
  }
  return undef;
}

# The following function does exactly what it sounds like *grin*
sub atomic_number_to_label($) {

  my($num) = shift;

  unless (defined($num)) {
  die "Undefined first argument passed to atomic_number_to_label()";
  }

  if ($num < 1 or $num > 103) {
  die "Illegal atomic number $num passed to atomic_number_to_label()";
  }

  if ($num == 1) { return "H" }
  elsif ($num == 2) { return "He" }
  elsif ($num == 3) { return "Li" }
  elsif ($num == 4) { return "Be" }
  elsif ($num == 5) { return "B" }
  elsif ($num == 6) { return "C" }
  elsif ($num == 7) { return "N" }
  elsif ($num == 8) { return "O" }
  elsif ($num == 9) { return "F" }
  elsif ($num == 10) { return "Ne" }
  elsif ($num == 11) { return "Na" }
  elsif ($num == 12) { return "Mg" }
  elsif ($num == 13) { return "Al" }
  elsif ($num == 14) { return "Si" }
  elsif ($num == 15) { return "P" }
  elsif ($num == 16) { return "S" }
  elsif ($num == 17) { return "Cl" }
  elsif ($num == 18) { return "Ar" }
  elsif ($num == 19) { return "K" }
  elsif ($num == 20) { return "Ca" }
  elsif ($num == 21) { return "Sc" }
  elsif ($num == 22) { return "Ti" }
  elsif ($num == 23) { return "V" }
  elsif ($num == 24) { return "Cr" }
  elsif ($num == 25) { return "Mn" }
  elsif ($num == 26) { return "Fe" }
  elsif ($num == 27) { return "Co" }
  elsif ($num == 28) { return "Ni" }
  elsif ($num == 29) { return "Cu" }
  elsif ($num == 30) { return "Zn" }
  elsif ($num == 31) { return "Ga" }
  elsif ($num == 32) { return "Ge" }
  elsif ($num == 33) { return "As" }
  elsif ($num == 34) { return "Se" }
  elsif ($num == 35) { return "Br" }
  elsif ($num == 36) { return "Kr" }
  elsif ($num == 37) { return "Rb" }
  elsif ($num == 38) { return "Sr" }
  elsif ($num == 39) { return "Y" }
  elsif ($num == 40) { return "Zr" }
  elsif ($num == 41) { return "Nb" }
  elsif ($num == 42) { return "Mo" }
  elsif ($num == 43) { return "Tc" }
  elsif ($num == 44) { return "Ru" }
  elsif ($num == 45) { return "Rh" }
  elsif ($num == 46) { return "Pd" }
  elsif ($num == 47) { return "Ag" }
  elsif ($num == 48) { return "Cd" }
  elsif ($num == 49) { return "In" }
  elsif ($num == 50) { return "Sn" }
  elsif ($num == 51) { return "Sb" }
  elsif ($num == 52) { return "Te" }
  elsif ($num == 53) { return "I" }
  elsif ($num == 54) { return "Xe" }
  elsif ($num == 55) { return "Cs" }
  elsif ($num == 56) { return "Ba" }
  elsif ($num == 57) { return "La" }
  elsif ($num == 58) { return "Ce" }
```

```
  elsif ($num == 59) { return "Pr" }
  elsif ($num == 60) { return "Nd" }
  elsif ($num == 61) { return "Pm" }
  elsif ($num == 62) { return "Sm" }
  elsif ($num == 63) { return "Eu" }
  elsif ($num == 64) { return "Gd" }
  elsif ($num == 65) { return "Tb" }
  elsif ($num == 66) { return "Dy" }
  elsif ($num == 67) { return "Ho" }
  elsif ($num == 68) { return "Er" }
  elsif ($num == 69) { return "Tm" }
  elsif ($num == 70) { return "Yb" }
  elsif ($num == 71) { return "Lu" }
  elsif ($num == 72) { return "Hf" }
  elsif ($num == 73) { return "Ta" }
  elsif ($num == 74) { return "W" }
  elsif ($num == 75) { return "Re" }
  elsif ($num == 76) { return "Os" }
  elsif ($num == 77) { return "Ir" }
  elsif ($num == 78) { return "Pt" }
  elsif ($num == 79) { return "Au" }
  elsif ($num == 80) { return "Hg" }
  elsif ($num == 81) { return "Tl" }
  elsif ($num == 82) { return "Pb" }
  elsif ($num == 83) { return "Bi" }
  elsif ($num == 84) { return "Po" }
  elsif ($num == 85) { return "At" }
  elsif ($num == 86) { return "Rn" }
  elsif ($num == 87) { return "Fr" }
  elsif ($num == 88) { return "Ra" }
  elsif ($num == 89) { return "Ac" }
  elsif ($num == 90) { return "Th" }
  elsif ($num == 91) { return "Pa" }
  elsif ($num == 92) { return "U" }
  elsif ($num == 93) { return "Np" }
  elsif ($num == 94) { return "Pu" }
  elsif ($num == 95) { return "Am" }
  elsif ($num == 96) { return "Cm" }
  elsif ($num == 97) { return "Bk" }
  elsif ($num == 98) { return "Cf" }
  elsif ($num == 99) { return "Es" }
  elsif ($num == 100) { return "Fm" }
  elsif ($num == 101) { return "Md" }
  elsif ($num == 102) { return "No" }
  elsif ($num == 103) { return "Lr" }
  else { return undef }

}

# The following function does exactly what it sounds like.
sub atomic_label_to_number($) {

  my($label) = shift;

  unless (defined($label)) {
  die "Undefined first argument passed to atomic_label_to_number()";
  }
  if ($label eq "H") { return 1 }
  elsif ($label eq "He") { return 2 }
  elsif ($label eq "Li") { return 3 }
  elsif ($label eq "Be") { return 4 }
  elsif ($label eq "B") { return 5 }
  elsif ($label eq "C") { return 6 }
  elsif ($label eq "N") { return 7 }
  elsif ($label eq "O") { return 8 }
  elsif ($label eq "F") { return 9 }
  elsif ($label eq "Ne") { return 10 }
  elsif ($label eq "Na") { return 11 }
  elsif ($label eq "Mg") { return 12 }
  elsif ($label eq "Al") { return 13 }
  elsif ($label eq "Si") { return 14 }
  elsif ($label eq "P") { return 15 }
  elsif ($label eq "S") { return 16 }
  elsif ($label eq "Cl") { return 17 }
  elsif ($label eq "Ar") { return 18 }
  elsif ($label eq "K") { return 19 }
  elsif ($label eq "Ca") { return 20 }
  elsif ($label eq "Sc") { return 21 }
  elsif ($label eq "Ti") { return 22 }
  elsif ($label eq "V") { return 23 }
  elsif ($label eq "Cr") { return 24 }
  elsif ($label eq "Mn") { return 25 }
  elsif ($label eq "Fe") { return 26 }
  elsif ($label eq "Co") { return 27 }
  elsif ($label eq "Ni") { return 28 }
  elsif ($label eq "Cu") { return 29 }
  elsif ($label eq "Zn") { return 30 }
  elsif ($label eq "Ga") { return 31 }
  elsif ($label eq "Ge") { return 32 }
  elsif ($label eq "As") { return 33 }
  elsif ($label eq "Se") { return 34 }
  elsif ($label eq "Br") { return 35 }
  elsif ($label eq "Kr") { return 36 }
  elsif ($label eq "Rb") { return 37 }
  elsif ($label eq "Sr") { return 38 }
  elsif ($label eq "Y") { return 39 }
  elsif ($label eq "Zr") { return 40 }
  elsif ($label eq "Nb") { return 41 }
  elsif ($label eq "Mo") { return 42 }
  elsif ($label eq "Tc") { return 43 }
  elsif ($label eq "Ru") { return 44 }
  elsif ($label eq "Rh") { return 45 }
  elsif ($label eq "Pd") { return 46 }
  elsif ($label eq "Ag") { return 47 }
  elsif ($label eq "Cd") { return 48 }
  elsif ($label eq "In") { return 49 }
  elsif ($label eq "Sn") { return 50 }
  elsif ($label eq "Sb") { return 51 }
  elsif ($label eq "Te") { return 52 }
  elsif ($label eq "I") { return 53 }
  elsif ($label eq "Xe") { return 54 }
  elsif ($label eq "Cs") { return 55 }
  elsif ($label eq "Ba") { return 56 }
  elsif ($label eq "La") { return 57 }
```

```
  elsif ($label eq "Ce") { return 58 }
  elsif ($label eq "Pr") { return 59 }
  elsif ($label eq "Nd") { return 60 }
  elsif ($label eq "Pm") { return 61 }
  elsif ($label eq "Sm") { return 62 }
  elsif ($label eq "Eu") { return 63 }
  elsif ($label eq "Gd") { return 64 }
  elsif ($label eq "Tb") { return 65 }
  elsif ($label eq "Dy") { return 66 }
  elsif ($label eq "Ho") { return 67 }
  elsif ($label eq "Er") { return 68 }
  elsif ($label eq "Tm") { return 69 }
  elsif ($label eq "Yb") { return 70 }
  elsif ($label eq "Lu") { return 71 }
  elsif ($label eq "Hf") { return 72 }
  elsif ($label eq "Ta") { return 73 }
  elsif ($label eq "W") { return 74 }
  elsif ($label eq "Re") { return 75 }
  elsif ($label eq "Os") { return 76 }
  elsif ($label eq "Ir") { return 77 }
  elsif ($label eq "Pt") { return 78 }
  elsif ($label eq "Au") { return 79 }
  elsif ($label eq "Hg") { return 80 }
  elsif ($label eq "Tl") { return 81 }
  elsif ($label eq "Pb") { return 82 }
  elsif ($label eq "Bi") { return 83 }
  elsif ($label eq "Po") { return 84 }
  elsif ($label eq "At") { return 85 }
  elsif ($label eq "Rn") { return 86 }
  elsif ($label eq "Fr") { return 87 }
  elsif ($label eq "Ra") { return 88 }
  elsif ($label eq "Ac") { return 89 }
  elsif ($label eq "Th") { return 90 }
  elsif ($label eq "Pa") { return 91 }
  elsif ($label eq "U") { return 92 }
  elsif ($label eq "Np") { return 93 }
  elsif ($label eq "Pu") { return 94 }
  elsif ($label eq "Am") { return 95 }
  elsif ($label eq "Cm") { return 96 }
  elsif ($label eq "Bk") { return 97 }
  elsif ($label eq "Cf") { return 98 }
  elsif ($label eq "Es") { return 99 }
  elsif ($label eq "Fm") { return 100 }
  elsif ($label eq "Md") { return 101 }
  elsif ($label eq "No") { return 102 }
  elsif ($label eq "Lr") { return 103 }
  else { return undef }

}

# End new(ish) functions, begin dinosaurs.

# The following function parses the file implied by the name of the
# job passed to it in it's sole argument (the input file) to see if the
# job actually completed.
sub is_finished {
  my($infile_name) = shift(@_);

  unless (defined $infile_name) {
    return 0;
  }

  # If someone passes a bare filename, the following doesn't work so well,
  # so we'll prepend the current directory to the path.
  unless ( $infile_name =~ m|/| ) { $infile_name = "./$infile_name"; }

  my(@scratch_list) = split("\/",$infile_name);
  my($in_base_name) = pop(@scratch_list);
  my($path) = join("\/", @scratch_list);
  @scratch_list = split("\\.", $in_base_name);
  $in_base_name = $scratch_list[0];

  # Now, we have enough information to open the correct .log file, which
  # is assumed to be in the working directory (path).  Note:  only if
  # the file exists!
  unless (-e "$path/${in_base_name}.log") { return 0; }
  open (WORK_FILE, "<$path/${in_base_name}.log") or my_warn("Could not " .
    "open $path/${in_base_name}.log for reading is is_finished()") and
    return 0;

  # Zoom to near the end of the file
  seek(WORK_FILE, -36, 2);
  my($line) = <WORK_FILE>;
  chomp($line);
  close(WORK_FILE);
  # Note that gaussian _always_ (as far as I can tell) finishes it's file
  # with the same text (therefore the -36 byte offset).
  if ($line ne " Normal termination of Gaussian 98\.") {
    my_warn("Job $path/${in_base_name}.log does not appear to be " .
        "finished, the last line was: $line");
    return 0;
  } else {
    return 1;
  }
}

# The following function simply reads the first geometry encountered in the
# logfile, and returns it as a list, such that each member of the list
# represents an atom, and the individual members are:
# @molecule[atom#][0] = <label>
# @molecule[atom#][1] = <x coordinate>
# @molecule[atom#][2] = <y coordinate>
# @molecule[atom#][3] = <z coordinate>
sub read_first_geometry($) {
  my($filename) = shift;
  my(@molecule);

  open(TMP, "<$filename") or
    die "Unable to open $filename for reading in read_first_geometry()";
  seek(TMP, 0, 0);

  my($found_start) = 0;
  my($i) = 1;
  while ( $line = <TMP> ) {
    if ( $line =~
      /^\s*(\w+)\s+(-?\d+\.\d*)\s+(-?\d+\.\d*)\s+(-?\d+\.\d*)/ ) {
      $found_start = 1;
      # Save this line, and all of the rest we come into
      $molecule[$i][0] = $1;
      $molecule[$i][1] = $2;
      $molecule[$i][2] = $3;
      $molecule[$i][3] = $4;
      $i++;
    } else {
      if ($found_start) { last }
    }
  }
  close(TMP);
  return @molecule;
}

# The following function extracts chelpg charges from the provided
# file.  It is the responsibility of the calling program to make
# certain the file is readable, etc.  This (g98) implementation will
# also guarantee that the file is a .log file, but other
# implementations may have to actually change the suffix if the local
# ab initio program uses a different one.

sub extract_chelpg_charges($) {
  my($filename) = shift;
  die "Illegal filename $filename in extract_chelpg_charges()"
    unless ( $filename =~ m/\.log$/ );

  open (INP, "<$filename") or die "Unable to open $filename in " .
    "extract_chelpg_charges()";

  my($line) = "";
  seek("INP", 0, 2);
  my($lastpos) = tell("INP");
  # Back it up a few bytes
  seek("INP", -1000, 1);

  # And throw away the partial line
  <INP>;

  until ( $line =~ /^\s*Charges from ESP fit,/ ) {
    if ( tell("INP") >= $lastpos ) {
      $lastpos -= 1000;
      if ( $lastpos == -1000 ) {
        die "Could not find chelpg section in input file $filename " .
          "in extract_chelpg_charges()";
      } elsif ( $lastpos < 0 ) {
        $lastpos = 0;
        seek("INP", 0, 0);
        next;
      }
      seek("INP", $lastpos, 0);
      <INP>;
      $lastpos = tell("INP");

      # And increment the position (by 1) so we will read the full line
      # we discarded
      $lastpos++;

      # And back up again
      seek("INP", -1000, 1);
      <INP>;
    }
    $line = <INP>;
    unless ( $line ) { print "\n\ntrouble coming\n\n"; }
  }

  # Now just get the charges:
  my(@return_list) = ();
  while ( ($line = <INP>) !~ /^\s*----------------------/ ) {
    if ( $line =~ /^\s*[\d]+\s+[\w]+\s+([+-]?\d+\.\d+)/ ) {
      push(@return_list, $1);
    }
  }
  return @return_list;
}


# And ... make this library return true
1;
```

local_functions.pl

```
# This file contains functions used in perl programs that are local to
# the environment we're using, such as functions to submit to dqs, and
# check for the least loaded server, etc.

# The following function queries (via qstat -f) the loads of the current
# user, and either returns the name of the machine that is most suitable
# for running a new job, or the empty string if no such machine is found.
sub check_system_resources {
  my($user) = shift(@_);
  my(@hosts) = @_;
  if ($#hosts < 1) { return "" }
  open(LOADS, "ssh $hosts[0] qstat -f |") or die
      "Could not get a load report from the system in " .
      "check_system_resources() ... exiting\n";

  my(@load_list) =  <LOADS>;
  close(LOADS);
  chomp(@load_list);
  my($current_job_count) = 0;

  $current_job_count = grep(/^\s+$user/, @load_list);

  if ($current_job_count > 8) { return ""; }
```

```
  @load_list = grep(/UP\s*/, @load_list);
  my($least_loaded) = "";
  my($lowest_load) = "1000";

    # The funky regexp in the next section basically extracts the hostname,
    # job #/# values, and the load from a line that looks like:
    #bandersnatch                    batch         0/3   2.00  er      UP

  for (my($i) = 0; $i <= $#load_list; $i++ ) {
    if ( $load_list[$i] =~
      /^(\w+)\s+\w+\s+(\d+)\/(\d+)\s+([0-9]+\.?[0-9]+)/) {
      if ( $4 <= $lowest_load && $2 < $3 ) {
        $least_loaded = $1;
        $lowest_load = $4;
      }
    }
  }
  return $least_loaded;
}

# The following function queries the server provided in the command line
# about whether or not the assigned program is running or not.
sub is_running {

  my($server) = shift(@_);
  my($job_name) = shift(@_);

  open(WORK_FILE, "ssh $server ps -ef |") or warn_out("Could not execute ".
    "\"ssh $server ps -ef |\"");
  my(@ps_list) = <WORK_FILE>;
  for(@ps_list) {
    if($_ =~ $job_name) { return 1; }
  }
  close(WORK_FILE);

  return 0;
}

# The following function submits the job to the local queing system, and
# could be as simple as submitting the job wholesale.  It takes a single
# argument which would be the command line to type in if you were to run
# the job manually.  Note that it also requires the full path name of the
# command, since it uses the first part to determine the working directory
# of the command.  It is assumed that the input file is the last argument
# (delineated by spaces) of the command line.  (This is necessary to be
# able to copy the file to the appropriate server).  Finally, this function
# will return a failure if it cannot start the job.  Some reasons might be:
# cannot create the necessary file in a unique directory, etc.
sub submit_job {

  my($user) = shift(@_);
  my($host) = shift(@_);
  my($cmd_line) = shift(@_);

  return;
}

# And ... make this library return true
1;
```

LINALG.pm

```
# This file is meant to be 'require' ed by any perl programs that need
# to do vector arithmetic.  It is invaluable for doing things like dihedrals,
# etc.

# Ok, this is my first attempt at being a real, modern module.

# BEGIN { $Exporter::Verbose=1 }

package LINALG;
require Exporter;

our @ISA    = qw(Exporter);
our @EXPORT    = qw();
our @EXPORT_OK  = qw(mat_print mat_add mat_sub mat_scalar_mult mat_mult
                    mat_trans mat_trace  mat_pow mat_inv is_square
                    mat_copy mat_det mat_cofac mat_adjoint
                    mat_get_identity v_add v_sub v_scalar_len v_scalar_mult
                    v_dot_prod v_proj v_perp v_norm v_x3 c_get_transition_mat
          c_orthog_transition_mat3 c_norm_transistion_mat
          c_orthnorm_transistion_mat3 c_solve_sys_lin_eq
          c_get_vec_in_basis c_transform c_get_dihedral
                    acos $PI
                    );
our $VERSION  = 0.01;
our %EXPORT_TAGS =  (
          basic  => [ qw(mat_print mat_add mat_sub
                                   mat_scalar_mult mat_mult mat_trans
                                   mat_trace mat_pow mat_inv mat_copy
                                   mat_det mat_cofac mat_adjoint
                   mat_get_identity v_add v_sub
                                   v_scalar_len
                   v_scalar_mult v_dot_prod v_proj
                   v_perp v_norm v_x3
                   c_get_transition_mat
                   c_orthog_transition_mat3
                   c_norm_transistion_mat
                   c_orthnorm_transistion_mat3
                   c_solve_sys_lin_eq
                   c_get_vec_in_basis c_transform
                   c_get_dihedral acos $PI) ],
         );

use POSIX;

# Prototypes (I learned the hard way that these functions don't know
# about the prototypes until they actually need them, so sometimes they
# guess wrong and pass a list instead of a reference.

# Constants and convenience functions
BEGIN {
  $PI = acos(-1);
}

# Matrix functions:  When these request a list, they require a list of
# references to lists.
sub mat_inv(\@);
sub mat_print (\@);
sub mat_dump (\@);
sub mat_add (\@\@);
sub mat_sub (\@\@);
sub mat_scalar_mult ($\@);
sub mat_mult (\@\@);
sub mat_trans(\@);
sub is_square(\@);
sub mat_trace (\@);
sub mat_pow ($\@);
sub mat_inv(\@);
sub is_proper(\@);
sub is_regular(\@);
sub mat_size(\@);
sub mat_copy(\@);
sub mat_det(\@);
sub mat_cofac($$\@);
sub mat_adjoint(\@);
sub mat_get_identity($);

# Vector functions
sub v_add(\@\@);
sub v_sub(\@\@);
sub v_scalar_len(\@);
sub v_scalar_mult($\@);
sub v_dot_prod(\@\@);
sub v_proj(\@\@);
sub v_perp(\@\@);
sub v_norm(\@);
sub v_x3(\@\@);

# Complex functions (not using complicated numbers, but having a level
# of complexity beyond primitive vector and matrix operations.

# WARNING!  All mentions of basis (for coordinate systems) is _rowwise_.
# This stands against some texts that I have seen, but also significantly
# simplifies creation of basis, both for the module, and for the user.

sub c_get_transition_mat(\@\@;$$$);
sub c_orthog_transition_mat3(\@);
sub c_norm_transistion_mat(\@);
sub c_orthnorm_transistion_mat3(\@);
sub c_solve_sys_lin_eq (\@\@);
sub c_get_vec_in_basis(\@\@);
sub c_transform(\@\@);
sub c_get_dihedral(\@);

# The majority of these functions will _not_ check to make certain the
# matrix is properly formatted, and regular.  They will assume things
# are fine.  If the user finds errors, they are encouraged to us
# mat_print on the questionable matrix to find out what's wrong.

# This function simply does a sort of 'pretty print' of the supplied
# matrix.  If passed a scalar, it simply prints that.
sub mat_print (\@) {
  my($matref) = shift;

  my($rowsize) = $#$matref;
  my($colsize) = $#{@$matref[0]};

  unless(is_proper(@$matref)) {
    print "Error:  Matrix is not properly formatted, or it may " .
      "be empty\n";
    return;
  }
  unless(is_regular(@$matref)) {
    print "Warning:  Matrix is not regular\n";
  }

  foreach(@$matref) {
    foreach(@{$_}) {
      printf("%-10.4G", $_);
    }
    print "\n";
  }

  if (is_regular(@$matref)) {
  print "That was a " . ($rowsize + 1) . " by " .
    ( $colsize + 1) . " matrix\n";
  }

  return;
}

# The following function does no error checking, but simply dumps the
# contents of the matrix, in the way it expects it to be formatted
sub mat_dump (\@) {

  my($mat) = shift;

  print "Dumping contents of matrix:\n";
  foreach(@$mat) {
    print join(" ", @{$_}) . "\n";
  }
  print "Dump complete\n";

  return;
}

# The following function returns an identity matrix, in the dimension
# requested with it's sole argument.  It will be 1 based, so 3 dimensional
# is 3 dimensional *grin*.  It may be useful for transformation of
# coordinate system from a standard basis to an arbitrary basis
```

```
sub mat_get_identity($) {
  my($dim) = shift;
  my($i, $j, @mat);

  $dim--;
  for $i ( 0 .. $dim ) {
    for $j ( 0 .. $dim ) {
      $mat[$i][$j] = ($i == $j ? 1 : 0) ;
    }
  }
  return @mat;
}

# The following function does an add operation to the two matrices
# provided.  If the matrices are not identical in size, it returns
# an empty list, as this situation is not defined for vector addition.  If
# you simply wish to add vectors, supply 1 x n or n x 1 matrices.
sub mat_add (\@\@) {
  my($mat1) = shift;
  my($mat2) = shift;
  my(@retmat);
  my($i, $j);

  unless( is_regular(@$mat1) and is_regular(@$mat2) ) {
    return undef;
  }
  unless(mat_size(@$mat1) eq mat_size(@$mat2)) {
    return undef;
  }

  for $i (0 .. $#$mat1 ) {
    for $j (0 .. $#{@$mat1[0]}) {
      $retmat[$i][$j] = $$mat1[$i][$j] + $$mat2[$i][$j];
    }
  }

  return @retmat;
}

# The following function subtracts the second matrix from the first
# If the matrices are not identical in size, it returns
# an empty list, as this situation is not defined for vector addition.  If
# you simply wish to add vectors, supply 1 x n or n x 1 matrices.
sub mat_sub (\@\@) {
  my($mat1) = shift;
  my($mat2) = shift;
  my(@retmat);
  my($i, $j);

  unless( is_regular(@$mat1) and is_regular(@$mat2) ) {
    return undef;
  }
  unless(mat_size(@$mat1) eq mat_size(@$mat2)) {
    return undef;
  }

  for $i (0 .. $#$mat1 ) {
    for $j (0 .. $#{@$mat1[0]}) {
      $retmat[$i][$j] = $$mat1[$i][$j] - $$mat2[$i][$j];
    }
  }

  return @retmat;
}

# The following function does a scalar multiplication of the matrix
# provided in the second argument by the scalar supplied in the first.
sub mat_scalar_mult ($\@) {
  my($fact) = shift;
  my($mat) = shift;
  my($i, $j);
  my(@retmat);

  for $i (0 .. $#$mat ) {
    for $j (0 .. $#{@$mat[0]}) {
      $retmat[$i][$j] = $fact * $$mat[$i][$j];
    }
  }
  return @retmat;
}

# The following function does a matrix multiplication of two matrices.
# It does check that the sizes of the matrices are correct.
sub mat_mult (\@\@) {
  my($mat1) = shift;
  my($mat2) = shift;
  my($i, $j, $k);
  my(@retmat);

  unless(is_regular(@$mat1) and is_regular(@$mat2)) { return undef }
  if ( $#{@$mat1[0]} != $#$mat2 ) { return undef }

  # And on with the multiplication
  for $i (0 .. $#$mat1 ) {
    for $j (0 .. $#{@$mat2[0]}) {
      for $k ( 0 .. $#{@$mat1[0]} ) {
        $retmat[$i][$j] += $$mat1[$i][$k] * $$mat2[$k][$j];
      }
    }
  }
  return @retmat;
}

# The following function returns the transpose of a matrix.  It turns out
# to be a bit funky to work with a non-regular matrix, so we'll check
# for that in the beginning.
sub mat_trans(\@) {
  my($mat) = shift;
  my($i, $j);
  my(@retmat);

  unless(is_regular(@$mat)) { return undef }

  for $i (0 .. $#$mat ) {
    for $j (0 .. $#{@$mat[0]}) {
      $retmat[$j][$i] = $$mat[$i][$j];
    }
  }
  return @retmat;
}

# The following function simply tests to see if the matrix is square
# (which implies regularity), and returns a 0 or 1 depending on the result.
sub is_square(\@) {
  my($mat) = shift;
  unless(is_regular(@$mat)) { return 0 }
  unless($#$mat == $#{@$mat[0]} ) { return 0 }

  return 1;
}

# The following function returns the trace of the matrix, the matrix must
# be square for this function to make sense.  The trace is simply the
# sum of the diagonal elements.  It returns undef on an error.
sub mat_trace (\@) {
  my($mat) = shift;
  my($i);
  my($trace) = 0;

  unless(is_square(@$mat)) { return undef; }
  for $i (0 .. $#mat ) {
    $trace += $$mat[$i][$i];
  }
  return $trace;
}

# The following function simply returns the matrix raised to the first
# argument's power.
sub mat_pow ($\@) {
  my($pow) = shift;
  my($mat) = shift;
  my(@retmat, @work_mat);
  my($i, $j);

  unless(is_square(@$mat)) { return undef }

  if ($pow == 0) {
    for $i ( 0 .. $#$mat ) {
      for $j ( 0 .. $#$mat ) {
        $retmat[$i][$j] = ($i == $j ? 1 : 0) ;
      }
    }
    return @retmat;
  }

  if ($pow < 0) {
    @work_mat = mat_copy(@$mat);
    @work_mat = mat_inv(@work_mat);
    @retmat = mat_copy(@work_mat);
    unless ($retmat[0]) { return undef }
    $pow *= -1;
  } else {
    @retmat = mat_copy(@$mat);
    @work_mat = mat_copy(@$mat);
  }

  for $i ( 2 .. $pow ) {
    @retmat = mat_mult(@retmat, @work_mat);
  }
  return @retmat;
}

# The following function may be the trickiest one by far.  It gives the
# inverse of the matrix passed to it.  If it is found in the process that
# the matrix is not invertible, it returns undef.  Note that it also returns
# undef if the matrix is not square - to disambiguate these two cases,
# check the matrix yourself with is_square(), or just mat_print() it.
sub mat_inv(\@) {
  my($orig_mat) = shift;
  my(@retmat);
  my($i, $j, $k);
  my(@vec);

  unless(is_square(@$orig_mat)) { return undef }

  # We need to copy the matrix, since we'll be mangling it
  my(@mat) = mat_copy(@$orig_mat);

  # Format our return matrix
  for $i ( 0 .. $#mat ) {
    for $j ( 0 .. $#mat ) {
      $retmat[$i][$j] = ($i == $j ? 1 : 0) ;
    }
  }

  # And begin the work.  Note that any transformation must be carried out
  # on both @mat and @retmat.  The algorithm is as follows:
  # 1) Do as many times as there are elements (in any row or column,
  #    since it's square) (This is $i);
  #  2) Sort the rows such that no row has fewer leading zeroes
  #      than the previous row.
  #  3) Divide the $ith row by it's first element
  #   4) Do from $i + 1 to as many elements in the matrix ($j) :
  #    5) For the element in this row, and $i(th) position, subtract
  #       that many times the $i(th) row, and subsitute the result for
  #       the current row.
  # 6) If any row has more than 1 more zero (leading) than the
  #    previous row, the matrix is uninvertable, return undef
  #    NOTE:  This can only be done after the initial diagonalization
  #      is finished!
  # At this point in the algorithm, we should have a upper triangular
  # matrix, so we finish the top part.
  # 7) Do as many times as there are elements (in reverse order) ($i);
  #  8) Do from $i - 1 to 0 ($j)
  #  9) For the element in this row, and $i(th) position, subtract
  #       that many times the $i(th) row, and subsitute the result for
  #       the current row.
  #
```

```
  #
  # And there you have it ... let's get to work.

  for ($i = 0; $i <= $#mat; $i++) {
    # Sort the rows.  I pondered using perl's sort routine for this,
    # but we need to do the same action to rows in both matrices,
    # so it wasn't a good choice.
    OUTSIDE: for ($j = 0; $j <= $#mat; $j++) {
      for ($k = $j + 1; $k <= $#mat; $k++) {
        if(leading_count($mat[$j]) > leading_count($mat[$k]) ) {
          # Swap the rows
          @vec = @{$mat[$j]};   # Creates new row
          $mat[$j] = $mat[$k];  # Reuses old row
          $mat[$k] = [ @vec ];  # Creates new row

          # And do the same to the return matrix
          @vec = @{$retmat[$j]};
          $retmat[$j] = $retmat[$k];
          $retmat[$k] = [ @vec ];

          # Decrement $j, and jump back outside to try again
          $j--;
          next OUTSIDE;
        }
      }
    }
    # Divide the $i(th) row by it's first element;
    my($val);
    for ($j = 0; $j <= $#mat; $j++ ) {
      if ($mat[$i][$j] != 0) { $val = $mat[$i][$j]; last }
    }
    # Now, divide the appropriate rows in both matrices
    for ($j = 0; $j <= $#mat; $j++ ) {
      $mat[$i][$j] /= $val;
      $retmat[$i][$j] /= $val;
    }

    # Now, eliminate the members from the rows that follow
    $val = 0;
    for ($j = $i + 1; $j <= $#mat; $j++) {
      # Find this row's multiplier

      # Yes, it's time again, for important programming lessons from
      # Josh.  The reason I'm using @vec1 and @vec2 is that the
      # quantity @{$mat[$i]} is a proper array, not a matrix type.
      # The difference being that even a 1 row matrix has a reference
      # to a list as it's first argument.  Someday, these subtlties
      # will stop biting me.

      my(@vec1) = ([ @{$mat[$i]} ]);  # Outside ($i) row
      my(@vec2) = ([ @{$mat[$j]} ]);  # This ($j) row
      $val = $mat[$j][$i];

      @vec = mat_scalar_mult( $val, @vec1 );
      @vec = mat_sub( @vec2, @vec);
      $mat[$j] = $vec[0];

      # And do the same things to the return matrix
      @vec1 = ([ @{$retmat[$i]} ]);
      @vec2 = ([ @{$retmat[$j]} ]);

      @vec = mat_scalar_mult( $val, @vec1 );
      @vec = mat_sub( @vec2, @vec);
      $retmat[$j] = $vec[0];
    }
  }

  # Now we check the rows to see if we're uninvertable
  for $j ( 1 .. $#mat ) {
    if (leading_count($mat[$j]) > (leading_count($mat[$j - 1]) + 1) )
      { return undef }
  }

  # Ok, our matrix is now an upper triangular matrix, we need only
  # do the top half now.

  for ($i = $#mat; $i >= 0; $i--) {
    # We no longer need to sort the rows

    # Now, eliminate the members from the rows that follow
    $val = 0;
    for ($j = $i - 1; $j >= 0; $j--) {
      # Find this row's multiplier

      my(@vec1) = ([ @{$mat[$i]} ]);
      my(@vec2) = ([ @{$mat[$j]} ]);
      $val = $mat[$j][$i];

      @vec = mat_scalar_mult( $val, @vec1 );
      @vec = mat_sub( @vec2, @vec);
      $mat[$j] = $vec[0];

      # And do the same things to the return matrix
      @vec1 = ([ @{$retmat[$i]} ]);
      @vec2 = ([ @{$retmat[$j]} ]);

      @vec = mat_scalar_mult( $val, @vec1 );
      @vec = mat_sub( @vec2, @vec);
      $retmat[$j] = $vec[0];
    }

    # If we got this far, it's invertable, no need to check for it
  }
  return @retmat;
}

# This function returns the determinant of a matrix.  Algorithmically, it's
# quite similar to mat_inv.  It's a simple row reduction problem, but we need
# to keep track of the changes we make to the original matrix, so we
# can multiply the upper triangular matrix by the proper value.
sub mat_det(\@) {
  my($orig_mat) = shift;
  my($i, $j, $k);
  my(@vec);
  my($retval) = 1;

  unless(is_square(@$orig_mat)) { return undef }

  # We need to copy the matrix, since we'll be mangling it
  my(@mat) = mat_copy(@$orig_mat);

  # See the algorithm notes in the description of the function
  # mat_inv.  We'll work quite similarly, except we won't be multiplying
  # rows (to get them to 1), and we'll have to keep track of how
  # many times we swap rows.  Other than that, the solution is
  # simple.

  for ($i = 0; $i <= $#mat; $i++) {
    OUTSIDE: for ($j = 0; $j <= $#mat; $j++) {
      for ($k = $j + 1; $k <= $#mat; $k++) {
        if(leading_count($mat[$j]) > leading_count($mat[$k]) ) {
          # Swap the rows
          @vec = @{$mat[$j]};   # Creates new row
          $mat[$j] = $mat[$k];  # Reuses old row
          $mat[$k] = [ @vec ];  # Creates new row

          # We switched rows, so ...
          $retval *= -1;

          # Decrement $j, and jump back outside to try again
          $j--;
          next OUTSIDE;
        }
      }
    }

    # Now, eliminate the members from the rows that follow, this
    # maintains the value of the determinant
    my($val) = 0;
    for ($j = $i + 1; $j <= $#mat; $j++) {
      # Find this row's multiplier
      $val = $mat[$j][$i] / $mat[$i][$i];

      my(@vec1) = ([ @{$mat[$i]} ]);  # Outside ($i) row
      my(@vec2) = ([ @{$mat[$j]} ]);  # This ($j) row

      @vec = mat_scalar_mult( $val, @vec1 );
      @vec = mat_sub( @vec2, @vec);
      $mat[$j] = $vec[0];
    }
  }

  # Finally, the value of the determinant is simply the product of the
  # diagonal elements, times the $retval ( +-1 ) we've been keeping
  # track of when we swap rows.
  for ($i = 0; $i <= $#mat; $i++) {
    $retval *= $mat[$i][$i];
  }

  return $retval;
}

# The following function returns the cofactor of the matrix for the element
# given in the first two arguments, which are taken to be row, row.  Note
# that it is 0 based.
sub mat_cofac($$\@) {
  my($row) = shift;
  my($col) = shift;
  my($mat) = shift;
  my($retval) = ($row + $col) % 2 ? -1 : 1;

  unless(is_square(@$mat)) { return undef };

  my(@work_mat, $i, $j, $ri, $rj);   # $ri, and $rj are the 'real' indexes
                   # for the submatrix
  $ri = $rj = 0;
  for ( $i = 0, $ri = 0; $i <= $#$mat; $i++, $ri++ ) {
    if ( $i == $row ) { $ri--; next; }
    for ( $j = 0, $rj = 0; $j <= $#$mat; $j++, $rj++ ) {
      if ( $j == $col ) { $rj--; next }
      $work_mat[$ri][$rj] = $$mat[$i][$j];
    }
  }

  # We have our submatrix, now we simply find the determinant of it,
  # and multiply by the existing retval
  $retval *= mat_det(@work_mat);
  return $retval;

}

# The following function returns the adjoint matrix for the provided matrix
sub mat_adjoint (\@) {
  my($mat) = shift;

  my(@w_mat, $i, $j, $val);

  for $i ( 0 .. $#$mat ) {
    for $j ( 0 .. $#$mat ) {
      $w_mat[$i][$j] = mat_cofac($i, $j, @$mat);
    }
  }

  return mat_trans(@w_mat);
}

# The following function just counts the number of leading zeros in a
# reference to an array, and returns that number
sub leading_count (\@) {
  my($vec) = shift;
  my($count) = 0;

  foreach(@$vec) {
    unless ($_ == 0) { return $count }
    $count++;
  }
  return $count;
}
```

```perl
# The following function simply returns true or false, depending on whether
# the matrix is regular or not.  It is primarily intended for internal
# usage within this module.
sub is_regular(\@) {
  my($mat) = shift;

  my($col) = $#{@$mat[0]};
  foreach (@$mat) {
    if ( $#{$_} != $col ) {
      return undef;
    }
  }
  return 1;
}

# The following function makes certain the matrix is properly formatted,
# i.e., it has each member of the list as a reference to a list, and is
# not zero sized;
sub is_proper (\@) {

  my($mat) = shift;

  if($#$mat == -1 or $#{@$mat[0]} == -1) {
    return 0;
  }

  foreach(@$mat) {
    # Also, we need to make sure all of the members are references
    # to arrays
    if ( ref($_) ne "ARRAY" ) {
      return 0;
    }
  }
  return 1;
}

# The following function takes a matrix and returns the size in a string
# formatted as "<rows>x<columns>".  It uses the number of the last element
# as the size of the row or column.  It does no size checking, and returns
# The number of elements in the first row as the number of columns
sub mat_size(\@) {
  return "$#{$_[0]}x$#{${$_[0]}[0]}";
}

# This function makes a completely new copy of the matrix.  Just saying
# @blah = @mat copies the references into the list, and this can cause
# some somewhat subtle bugs.
sub mat_copy(\@) {
  my($mat) = shift;
  my(@retmat);

  foreach(@$mat) {
    push(@retmat, [@{$_}]);
  }
  return @retmat;

}

# The following function adds two vectors up to and including the last
# list index.  If one vector is longer than the other, the minimum
# length vector is calculated
sub v_add (\@\@) {
  my($vec1) = shift;
  my($vec2) = shift;
  my($dim) = $#$vec1 < $#$vec2 ?  $#vec1 : $#$vec2;
  my($i);
  my(@retvec);

  for $i ( 0 .. $dim ) {
    $retvec[$i] = $$vec1[$i] + $$vec2[$i];
  }
  return @retvec
}

# The following function subtracts the second vector from the first.
# If one vector is longer than the other, the minimum length vector
# is calculated
sub v_sub (\@\@) {

  my($vec1) = shift;
  my($vec2) = shift;
  my($dim) = $#$vec1 < $#$vec2 ?  $#vec1 : $#$vec2;
  my($i);
  my(@retvec);

  for $i ( 0 .. $dim ) {
    unless (defined $$vec1[$i]) {
    die "In v_sub:  The first vector passed has an undefined " .
        "value at index $i";
    }
    unless (defined $$vec2[$i]) {
    die "In v_sub:  The second vector passed has an undefined " .
        "value at index $i";
    }
    $retvec[$i] = $$vec1[$i] - $$vec2[$i];
  }
  return @retvec


}

# This function simply returns the length (in arbitrary space) of the
# vector in question
sub v_scalar_len (\@) {
  my($vec) = shift;
  my($sumsq);

  foreach (@$vec) {
    $sumsq += $_ * $_;
  }
  return sqrt($sumsq);
}

# The next function returns the result of scalar multiplication of the
# scalar and array passed to it
sub v_scalar_mult ($\@) {
  my($mult) = shift;
  my($vec) = shift;
  my(@retvec);

  foreach (@$vec) {
    push(@retvec, ( $_ * $mult) );
  }
  return @retvec;
}

# The next function simply return the dot product of two vectors.  If
# they are different sizes, only the minimum length is used
sub v_dot_prod (\@\@) {
  my($vec1) = shift;
  my($vec2) = shift;
  my($dim) = $#$vec1 < $#$vec2 ?  $#vec1 : $#$vec2;
  my($i);
  my($retval);

  for $i ( 0 .. $dim ) {
    $retval += $$vec1[$i] * $$vec2[$i];
  }
  return $retval;

}

# The following function finds the projection of the second vector along
# The first vector.  This order was chosen to mimic the linear algebraic
# expression proj(a)u, where a is the first vector, and u is the second
# Note:  This function will give goofy results if the vectors are not the
# same sizes ... don't abuse the functions!
sub v_proj(\@\@) {
  my($vec1) = shift;
  my($vec2) = shift;
  my($val, $len);

  $val = v_dot_prod(@$vec1, @$vec2);
  $len = v_scalar_len(@$vec1);
  if ( $len == 0 ) {
    my(@retvec);
    foreach(@$vec1) { push(@retvec, 0) }
    return @retvec;
  }
  $val = $val / ( $len * $len );
  return v_scalar_mult($val, @$vec1);

}

# This function uses the same protocol as v_proj()
sub v_perp(\@\@) {
  my($vec1) = shift;
  my($vec2) = shift;

  my(@w_vec);
  @w_vec = v_proj(@$vec1, @$vec2);
  return v_sub(@$vec2, @w_vec);
}

sub v_norm(\@) {
  my($vec) = shift;
  $factor = v_scalar_len(@$vec);
  if ( $factor == 0 ) {
    my(@retvec);
    foreach(@$vec) { push(@retvec, 0) }
    return @retvec;
  }
  $factor = 1 / $factor;

  return v_scalar_mult($factor, @$vec);
}

# The following function returns the cross product for the two vectors
# supplied in the arguments.  If either vector does not have exactly
# three members, it returns undef, as the definition of cross product that
# I have makes no sense in any other space
sub v_x3 (\@\@) {
  my($vec1) = shift;
  my($vec2) = shift;
  my(@retvec);

  unless ($#$vec1 == 2 and $#$vec2 == 2) { return undef }

  # I'll use no grace in returning this value, as it's a very specific
  # case, only applicable in three dimensions.
  $retvec[0] = $$vec1[1] * $$vec2[2] - $$vec2[1] * $$vec1[2];
  $retvec[1] = $$vec1[2] * $$vec2[0] - $$vec2[2] * $$vec1[0];
  $retvec[2] = $$vec1[0] * $$vec2[1] - $$vec2[0] * $$vec1[1];

  return @retvec;
}

# Begin more complex functions (not having to do with complex numbers).
# These functions are composite, and do more complex tasks than the
# primatives in the preview sections

# The following function returns a transition matrix from the coordinate
# system in the first argument, to the coordinates system in the second
# argument.  If the third (optional) is true, it uses an identity matrix
# for the from system instead of the one provided in the first arguement.
# If the fourth argument is true, it ortho-normalizes the to basis before
# proceeding.  If the fifth argument is true, it ortho-normalizes the to
# basis as well.  Note that it takes the coordinate systems in a row-wise
# fashion.  This means that the first row is the x (new or old) axis
# basis, the second row is the y axis basis, etc.  Note that some options
# only makes sense when we're in 3 dimensions, if we get a sensless option
# in the wrong dimensionality, we return undef.  It returns the transition
# matrix in the 'correct sense'.
sub c_get_transition_mat(\@\@;$$$) {
  my($mat1) = shift;
  my($mat2) = shift;
  my($use_identity) = shift;
```

```
  my($ortho_normalize_first) = shift;
  my($ortho_normalize_second) = shift;

  my(@from_mat, @to_mat);

  unless( is_square(@$mat1) and is_square(@$mat2) ) { return undef }
  if ( $#$mat1 != 2 and
    ($ortho_normalize_first or $ortho_normalize_second) ) { return undef }

  unless ( $#$mat1 == $#$mat2 ) { return undef }

  if($use_identity) {
    @from_mat = mat_get_identity($#$mat1 + 1);
  } else {
    @from_mat = mat_copy(@$mat1);
  }

  if($ortho_normalize_first) {
    @from_mat = c_orthnorm_transistion_mat3(@$mat1);
    unless($from_mat[0]) { return undef }
  } else {
    @from_mat = mat_copy(@$mat1);
  }

  if($ortho_normalize_second) {
    @to_mat = c_orthnorm_transistion_mat3(@$mat2);
    unless($to_mat[0]) { return undef }
  } else {
    @to_mat = mat_copy(@$mat2);
  }

  my($i, @work_mat, @retmat);

  for $i ( 0 .. $#from_mat ) {
    my(@this_vec) = @{$from_mat[$i]};
    $retmat[$i] = [ c_get_vec_in_basis(@this_vec, @to_mat) ];
  }

  @retmat = mat_trans(@retmat);
  return @retmat;
}

# The following function transforms the vector in the first argument
# based on the tranformation matrix provided in the second argument.  The
# function itself is quite simple, but takes advantage of preparing a
# tranform matrix ahead of time.
sub c_transform(\@\@) {
  my($vec) = shift;
  my($mat) = shift;

  my(@work_mat) = [ @$vec ];
  @work_mat = mat_trans(@work_mat);

  @work_mat  = mat_mult( @$mat, @work_mat );
  my(@retmat, $i);
  for $i ( 0 .. $#work_mat ) {
    $retmat[$i] = $work_mat[$i][0];
  }
  return @retmat;
}


sub c_orthog_transition_mat3(\@) {
  my($mat) = shift;
  my(@retmat);

  unless(mat_size(@$mat) eq "2x2") { return undef }

  @retmat = mat_copy(@$mat);
  $retmat[2] = [ v_x3(@{$retmat[0]}, @{$retmat[1]}) ];

  return @retmat;
}

sub c_norm_transistion_mat(\@) {
  my($mat) = shift;
  my(@retmat);
  foreach(@$mat) {
    push(@retmat, [ v_norm(@{$_}) ] );
  }
  return @retmat;
}

sub c_orthnorm_transistion_mat3(\@) {
  my($mat) = shift;
  my(@work_mat) = c_orthog_transition_mat3(@$mat);
  return c_norm_transistion_mat(@work_mat);
}

# The following function solves the system of linear equations represented
# by AX = B.  An example follows:
# ax + by = n
# cx + dy = m
# So: A = | a b |   and B = | n |, and the function returns m.
#         | c d |           | m |
# Note that the second argument is taken as a list, not a matrix ( list of
# references ).  Also, this _only_ works for n equations in n unknowns,
# that is, the matrix A must be square, and invertable.
sub c_solve_sys_lin_eq (\@\@) {
  my($mat) = shift;
  my($vec) = shift;
  my($i, @retvec, @work_mat, @work_mat2, @work_vec);

  # If the matrix is not the same size as the vector, this is a
  # nonsensible request.
  unless ($#$mat == $#$vec) { return undef }

  @work_vec = [ @{$vec} ];
  @work_vec = mat_trans(@work_vec);

  @work_mat = mat_inv(@$mat);
  unless($work_mat[0]) { return undef } # We failed to invert
  @work_mat2 = mat_mult(@work_mat, @work_vec);

  @retvec = ();
  for $i ( 0 .. $#$vec ) {
    push (@retvec, $work_mat2[$i][0]);
  }
  return @retvec;
}

# This function returns the vector in the first argument transformed to
# the coordinate system specified in the second basis.  Note that the second
# basis is row-wise, such that the first row represents the first axis, etc.
sub c_get_vec_in_basis(\@\@) {
  my($vec) = shift;
  my($mat) = shift;
  my(@work_mat) = mat_trans(@$mat);

  return c_solve_sys_lin_eq(@work_mat, @$vec);
}

# This is the culmination of (almost) all of my reason for writing this
# library.  This function takes as it's sole argument a list (four members
# long) of four coordinates in three space.  The dihedral is as defined in
# the famous paper: W. Klyne and V. Prelog (Experientia, 1960, 16, 521-523).
# The first member of the dihedral is accepted to the closest atom to you.
sub c_get_dihedral(\@) {
  my($dihed) = shift;
  my(@work_vec);

  if ($#$dihed != 3) { return undef }

  my($i);
  for $i ( 0 .. 3 ) {
    if ($#{@$dihed[$i]} != 2) { return undef }
  }

  my(@p1, @p2, @p3, @p4);
  @p1 = @{@$dihed[0]};
  @p2 = @{@$dihed[1]};
  @p3 = @{@$dihed[2]};
  @p4 = @{@$dihed[3]};

  my(@pvec1, @pvec2, @mid);  # projection vectors
  @mid = v_sub(@p2, @p3);
  @work_vec = v_sub(@p1, @p2);
  @pvec1 = v_perp(@mid, @work_vec);

  @work_vec = v_sub(@p4, @p3);
  @pvec2 = v_perp(@mid, @work_vec);

  # We have our two projections, now, analyze the angle between them

  my($cos, @nmid, @ncross);
  $cos = v_dot_prod(@pvec1, @pvec2) /
        ( v_scalar_len(@pvec1) * v_scalar_len(@pvec2) );
  @ncross = v_x3(@pvec1, @pvec2);
  @ncross = v_norm(@ncross);
  @nmid = v_norm(@mid);

  # The normalized cross product of our projection vectors and the
  # normalized middle vector will now either be in the same direction or
  # in opposite directions, this is easy enough to figure out, and the
  # sign of the cross product determines if we report the actual angle,
  # or 2 PI - angle
  @work_vec = v_sub(@ncross, @nmid);
  my($len) = v_scalar_len(@work_vec);

  # The length should be either 0 or 2, but we'll allow for floating
  # point and roundoff error when we return the result.
  if ($len < 0.01) {
    return 2 * acos(-1) - acos($cos);       # acos(-1) = PI
  } else {
    return acos($cos);
  }
}

# And make the library return true
return 1;
```

NETFLOCK.pm

```
# This file will implement file locking over a NFS network, as the
# flock() family of functions does not handle file locking in this
# case.  Please note that the algorithms implemented here may not be
# 'foolproof', i.e., race conditions could keep two processes fighting
# over the same file for some time.  I'm not certain how to change
# this, except for perhaps making a random timeout when one process
# cannot get a lock.  Note that it will also leave lock files around
# when it exits.  As an 'alpha' attempt to remedy this situation, I
# will try to keep track of a locked files list, and unlink them all
# in an end block included in this module.  Note also that it would be
# better to have this module have it's own timeout setting, so files
# that appear to be locked 'forever' simply caused the calling program
# to die.  This may be implemented later.  Finally, note that this
# module traps all normal signals, so the END block is executed.  Note
# that it will not catch all signals, and if this functionality is
# needed later, it should be added.  If the program that uses this
# module needs to handle signals, it should install it's handlers
# _after_ using this module.

# Ok, this is my second attempt at being a real, modern module.

# BEGIN { $Exporter::Verbose=1 }

package NETFLOCK;
require Exporter;
use strict;

use sigtrap qw(die untrapped normal-signals);
```

```
our @ISA    = qw(Exporter);
our @EXPORT    = qw();
our @EXPORT_OK  = qw( nflock nfunlock );
our $VERSION  = 0.01;
our %EXPORT_TAGS =  (
            basic  => [ qw( nflock nfunlock ) ],
          );

local(%NETFLOCK::locked_file_list);

# Prototypes (I learned the hard way that these functions don't know
# about the prototypes until they actually need them, so sometimes they
# guess wrong and pass a list instead of a reference.
sub nflock($);
sub nfunlock($);
sub nflockcleanup();

sub nflock($) {

  my($filename) = shift;

  $NETFLOCK::locked_file_list{"$filename.nflock"} = 1;

  # The basic procedure here is somewhat straightforward.  In order to
  # get a lock, we first see if there's a file called
  # <filename>.nflock.  If there is, we simply go to sleep.  If
  # there's not, we write one ourselves, with our pid on the only
  # line.  We then sleep for 1/10 of a second, and check again for the
  # presense of the file.  If it's not there, we repeat.  If it is, we
  # check to see if the pid in the file is our own pid.  If not, we
  # repeat.  If it is, we grant the lock and return to the caller.

  while ( 1 ) {
  if (-e "$filename.nflock") {

    # See if it's our lock
    open(TMP, "<$filename.nflock") or die "Unable to open " .
    "$filename.nflock for reading in nflock()";
    my($lockpid) = <TMP>;
    chomp($lockpid);
    close(TMP);
    if ( $lockpid == $$) {
    # Give the lock and return
    return 1
    }

    # The rest of this is effectively an else section

    select(undef, undef, undef, 0.10);
    next;
  }

  # If we get here, the filename must not exist, create it
  open(TMP, ">$filename.nflock") or die "Unable to open $filename.nflock " .
    "for writing in nflock(), this is a critical error";
  print TMP "$$\n";
  close(TMP);

  # Sleep for a bit
  select(undef, undef, undef, 0.10);

  open(TMP, "<$filename.nflock") or die "Unable to open $filename.nflock " .
    "for reading in nflock(), this is a critical error";
  my($lockpid) = <TMP>;
  close(TMP);
  chomp($lockpid);

  if ( $lockpid == $$ ) {
    # We have our lock, return
    return 1;
  } else {
    # Sleep for a bit, and try again
    select(undef, undef, undef, 0.10);
  }
  }
}

# This function is significantly simpler, we simply delete the lock
# file, and return

sub nfunlock($) {

  my($filename) = shift;

  delete($NETFLOCK::locked_file_list{"$filename.nflock"});

  if (-e "$filename.nflock") {
  unlink("$filename.nflock") or
    die "Unable to unlink $filename.nflock in nfunlock()";
    return 1;
  }
}

# Release any files we locked ourselves.  Note that this may have the
# unwanted side effect of releasing files locked from another
# invocation of this library (if more than one program is using this
# library, and the same file), though if keeping track of our own
# files via the hash is working, then we should be ok.
sub nflockcleanup() {

  foreach (keys(%NETFLOCK::locked_file_list)) {
  unlink($_);
  }
}

END {
  nflockcleanup();
}

# And make the library return true
return 1;
```

qdb

Database programs

torsion_driver.pl

```
#!/usr/bin/perl -wT

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# Changes, 4-20-02.  The current version has some problems, in that it
# often does exactly one torsion, and then sleeps forever.  This means
# that it must be restarted daily in order to get the job done, and
# this is inacceptable.  In all honesty, the driver needs a nearly
# complete overhaul, but for now, I believe the best thing we can do
# is change the test for 'do we make another interation' to, instead
# of looking for the message file (which works fine, unless the daemon
# has been restarted), look for the message file, and also look to see
# if the last log file is a finished calculation.

# This program was originally copied from an intermediate version of
# qdb_input_server.pl, as it's design is similar in that it's a daemon,
# and it shares some other similarities.
# Here in the intro, I try to lay out all characteristics, responsibilities,
# and structure requirements for the program.
# Characteristics:
#  - This program is a daemon, and is meant to run constantly.
#     This daemon is expected to be running in many instances at once.
#  - It is able to recover from partial calculations regardless of
#     whether or not it is starting 'fresh'.
#  - The daemon will use signals and files for interprocess communication,
#    which is mainly limited to telling the input server it's done.
#   - The program will be reasonably secure, and since it has no sockets
#     to the world, the only damage a potentially malicious party could
#     do is limitied to what someone with shell access can do.  This means
#     that until the program has had a serious security audit, it should
#     not be called as a result of any external call (i.e., from a web
#     page, or whatnot).  Note that this doesn't mean I'll be ignoring
#     security.  Once the program is started, it will set it's own home
#     directory to it's working directory.
#  - A design goal (that I've largely been ignoring with my perl programs
#    until now) is to keep as much of the 'crunching' out of the main
#    package as possible.  This will add to the readability of main.
#  - All user functions must be prototyped
# Responsibilities:
#  - The program is responsible for doing geometry optimizations at
#    a series of dihedrals, with the value of the next dihedral depending
#    on the energies of all of the previous dihedrals.
#  - The program is responsible for informing it's parent when it is
#     finished with the torsion scan
# Structure:
#  - Initialization
#    - No notable steps here that I can think of.  After work began,
#      a fair number of steps were implemented
#  - Main loop (until is_scan_complete() )
#    - sleep
#    - when awakened:
#      - re-read all input/intermediate files
#      - calculate what torsion we need next
#      - submit the torsion
#   - After main loop
#    - Simply call finish_and_exit().  The details are outlined in that
#      function.

# Note added later.  This program should be using my own nflock for
# manipulating the que.  It should be changed at some point in the
# future.

BEGIN {
  # Since our own modules aren't properly installed, add to the INC
  # list at compile time
  push(@INC, "../perl_modules");
}

eval { require 5.6.1 }
  or die <<MESSAGE;
##################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
```

```perl
### information
####################################################################
MESSAGE

package main;

use strict;
use sigtrap 'handler', \&wake_up, "ALRM";
use sigtrap 'handler', \&handle_hup, "HUP";
use sigtrap 'handler', \&quit_now, "QUIT";

# Note:  We need to use Cwd _after_ POSIX, since POSIX also has a getcwd
# function, and that module's implementation suffers from taint checking
# problems within the module.  The symptom of this is that it's not even
# possible to make a call to getcwd without failing on a taint error within
# POSIX.pm
use POSIX;
use Cwd;
use Fcntl ':flock';       # import LOCK_* Constants

require "../perl_modules/local_functions.pl";

# Command line syntax checking;
if ( scalar(@ARGV) != 4 ) {
  die "Please call this program with four arguments.  The first is " .
  "the home directory ( in the qdb ) in which to do the torsion " .
    "scan, and the second and third are the two atoms for which " .
    "to build it.  The fourth is the pid for the prgram that should " .
      "be informed when the torsion is finished.  Exiting.\n";
}

unless ( -e $ARGV[0] and -r $ARGV[0] and -d $ARGV[0]) {
  die "$ARGV[0] is not a readable existing directory";
}

unless ( $ARGV[1] =~ /^[\d]+$/ ) {
  die "$ARGV[1] is not a  positiveinteger";
}

unless ( $ARGV[2] =~ /^[\d]+$/ ) {
  die "$ARGV[2] is not a positive integer";
}

unless ( $ARGV[3] =~ /^[\d]+$/ ) {
  die "$ARGV[3] is not a positive integer, which is what I expect for " .
  "a pid, if your system provides non-integral pids, please " .
    "correct $0";
}

# Before proceeding, clean up our environment so we can run external
# programs
# Is this necessary?
# require '../general/clean_environment.pl';
# full_env_clean();

# Function prototypes

sub print_molecule (@);
sub print_connectivity(%);
sub finish_and_exit(\%);
sub choose_next_dihedral(%);
sub get_angle_energy($\@);
sub count_valid_energies(@);
sub get_line_parameters($$$$);
sub submit_torsion_job($\@\@;$);
sub make_dihed_inp_file($$\@\@\@$);
sub norm_angle($);
sub norm_round_angle($);

# The following are prototyped to force perl to call them properly
# In ../perl_modules/local_functions.pl

# In ../perl_modules/${ab_initio_program}_functions.pl
sub get_optimized_structures($);
sub get_last_dihedral_and_energy($\@);
sub get_optimized_connectivity($);

# Global variables
my($got_alarm) = 1;
my($loop_forever) = 1;
my($debug) = 0;
my($starting_directory) = getcwd;
my($this_host) = $ENV{"HOSTNAME"};
my($ab_initio_program);
my($ab_initio_suffix);
my($working_directory);
my(@central_dihedral);
my(%connectivity);
my(@frozen_dihedrals);
my($callers_pid) = $ARGV[3];
my(%angnrg);          # Must be global for finish_and_exit()

# Here is where we get anything we need from .qdb_checkrc

require "../general/rc_file_handling.pl";

open("RCFILE", '<.qdb_checkrc') or die
  "Unable to open .qdb_checkrc ... exiting\n";

$ab_initio_program = read_scalar("RCFILE", "local_ab_initio_program");
defined($ab_initio_program) or die
  "Unable to find ab_initio_program in .qdb_checkrc file ... exiting\n";

$ab_initio_suffix = read_scalar("RCFILE", "ab_initio_suffix");
defined($ab_initio_suffix) or die
  "Unable to find ab_inition_suffix in .qdb_checkrc file ... exiting\n";

my($sampling_rate) = read_scalar("RCFILE", "torsion_sampling_rate");
defined($sampling_rate) or die
  "Unable to find torsion_sampling_rate in .qdb_checkrc file " .
  " ... exiting\n";

my($min_samp_angle) = read_scalar("RCFILE", "minimum_sampling_angle");
defined($min_samp_angle) or die
  "Unable to find minimum_sampling_angle in .qdb_checkrc file " .
  " ... exiting\n";
if ( $min_samp_angle < 1 ) {
  $min_samp_angle = 1;
}
# This allows angles of 1 to be scanned, and is the minimum

my($nrg_cutoff) = read_scalar("RCFILE", "torsion_energy_cutoff");
defined($nrg_cutoff) or die
  "Unable to find torsion_energy_cutoff in .qdb_checkrc file " .
  " ... exiting\n";

my($this_user) = read_scalar("RCFILE", "user_name");
defined($this_user) or die
  "Unable to find this_user in .qdb_checkrc file ... exiting\n";


close("RCFILE");

my($rcfilename) = getcwd . "/.qdb_checkrc";

# Reset the alarm, and start it
my($alarm_timeout) = 600; # How often do we do general checking of things?
              # Note that we should never actually have to
             # check anything with this timer, but this is
              # here to handle the odd case the I missed
              # some logic in receiving signals from other
              # sources

# Launder any variables giving us trouble here

# Launder starting directory.  It _must_ be the directory we started in,
# otherwise the program will die when it tries to read .qdb_checkrc.
($starting_directory) =
  $starting_directory =~ m%^([-./\\\w-]+)$%;
# %'s used since / is in the pattern.

# Launder $rcfilename.  If this variable is insecure, make_dihed_inp_file()
# may do a different gaussian job than we want *shrug*
($rcfilename) = $rcfilename =~ m%^([-./\\\w]+)%;

# Launder $this_host ... if someone messes with HOSTNAME in the environment,
# the only ill effect is that the proper host won't get a SIGALRM when the
# calculation is finished ... that's pretty innocuous, huh?
($this_host) = $this_host =~ m%([\w.]+)%;

($ab_initio_program) = $ab_initio_program =~ m/([\w]+)/;

# Launder callers pid.
($callers_pid) = $callers_pid =~ m/(\d+)/;

require "../perl_modules/${ab_initio_program}_functions.pl";
# The following module requires that the proper directory be added to
# @INC at copile time, see the BEGIN block.
use LINALG qw(:basic);

# Before entering the main loop, let's finish checking our command line
# options, initialize the fragment, and create our directory structure.
# if it's not already there.

my(@opt_struct) =
  get_optimized_structure("$ARGV[0]/Initial_optimization.log");

# Finally, make sure $ARGV[1] and $ARGV[0] are ok.  First, are they
# connected?  Second, are they small enough to be part of the molecule?
open(TMP, "<$ARGV[0]/Connectivity.raw") or
  die "Unable to open $ARGV[0]/Connectivity.raw for reading";
my(@work_list) = <TMP>;
close(TMP);
chomp(@work_list);
my($is_bonded) = 0;
foreach (@work_list) {
  if ($_ =~ /^$ARGV[1] $ARGV[2]/ or $_ =~ /^$ARGV[2] $ARGV[1]/) {
  $is_bonded = 1;
  last;
  }
}
unless ($is_bonded) {
  die "Atoms $ARGV[1] and $ARGV[2] are not bonded";
}

# Initialize our connectivity hash
foreach (@work_list) {
  my($atom1, $atom2, undef) = split(" ", $_);
  push(@{$connectivity{$atom1}}, $atom2);
  push(@{$connectivity{$atom2}}, $atom1);
}

# And make the lists at each atom sorted
foreach (keys(%connectivity)) {
  @work_list = sort {$a<=>$b} @{$connectivity{$_}};
  $connectivity{$_} = [ @work_list ];
}

# Let's get our central dihedral
$central_dihedral[1] = $ARGV[1] < $ARGV[2] ? $ARGV[1] : $ARGV[2];
$central_dihedral[2] = $ARGV[1] < $ARGV[2] ? $ARGV[2] : $ARGV[1];

# If a terminal atoms is requested, remove our current directory, and
# then just exit with the die statement, so the calling program knows
# it goofed up (or the user, for that matter).
@work_list = @{$connectivity{$central_dihedral[1]}};
if ( $#work_list < 1 ) {
  die "Terminal atom ($central_dihedral[1]) received, this is a " .
  "critical error, cannot continue.  The directory was" .
    "$ARGV[0], the atoms were $ARGV[1] and $ARGV[2].  Somebody " .
    "passed me the wrong atoms!";
}

$central_dihedral[0] = ( $central_dihedral[2] != $work_list[0] ?
            $work_list[0] :
            $work_list[1] ) ;

@work_list = sort {$a<=>$b} @{$connectivity{$central_dihedral[2]}};
```

```
if ( $#work_list < 1 ) {
  die "Terminal atom ($central_dihedral[2]) received, this is a " .
  "critical error, cannot continue.  The directory was" .
    "$ARGV[0], the atoms were $ARGV[1] and $ARGV[2].  Somebody " .
    "passed me the wrong atoms!";
}

$central_dihedral[3] = ( $central_dihedral[1] != $work_list[0] ?
             $work_list[0] :
             $work_list[1] ) ;

# And make sure we were able to complete the dihedral.
unless ( defined($central_dihedral[0]) and
     defined($central_dihedral[3])  ) {
  die "Unable to complete dihedral (We were given a terminal atom?)";
}

# Create our working directory (if needed) and chdir into it.
if ( $ARGV[1] < $ARGV[2] ) {
  $working_directory = "$ARGV[0]/torsion_$ARGV[1]-$ARGV[2]";
} else {
  $working_directory = "$ARGV[0]/torsion_$ARGV[2]-$ARGV[1]";
}

($working_directory) = $working_directory =~ m%^([-.\\/\w]+)$%;

unless ( -e $working_directory ) {
  mkdir($working_directory, 0775) or
  die "Unable to create $working_directory";
}

# And go there.
chdir($working_directory);

########################################################################
######            Troubleshooting select bit here.             #########
########################################################################
print "Running in $working_directory, under runlog.txt\n";

open(BLAH, ">runlog.txt") or die "Unable to open logfile for writing";
select(BLAH);
$| = 1;
my($now) = localtime;
print $now;
print "\nServer sending output to log files\n";


# If we're already running here, we need to exit immediately.
if ( -e "driver_is_running" ) {
  die "Unable to start daemon, within this directory (" . getcwd .
  "), there is a driver_is_running file, which indicates that " .
  "there is either another instance of ourselves running, or " .
  "that the driver crashed previously.  If the driver is not " .
  "active for this directory, please remove the file, and try " .
  "again.";
} else {
  open(TMP, ">driver_is_running") or
  die "Unable to open critical file for writing";
  close(TMP);
}

# We may be able to bail out early if it appears this calculation has
# already been done.
if ( -e "angle_vs_energy" ) {
  print "Directory: $working_directory already finished, exiting\n";
  unlink "driver_is_running";
  exit;
  %angnrg = (); finish_and_exit(%angnrg);
}

# Initialize our frozen_dihedrals list.  For thoroughness, we'll
# freeze _all_ dihedrals with a given central pair of atoms.

open(TMP, "<../Frozen_bonds") or
  die "Unable to open ../Frozen_bonds for reading";
@work_list = <TMP>;
close(TMP);
chomp(@work_list);

@frozen_dihedrals = ();
foreach (@work_list) {
  my($atom1, $atom2) = split(" ", $_);
  foreach ( @{ $connectivity{$atom1} } ) {
  my($leftside) = $_;
  if ($leftside == $atom2) {
    next;
  }
  foreach ( @{ $connectivity{$atom2} } ) {
    my($rightside) = $_;
    if ($rightside == $atom1) {
    next;
    }
    # Finally, we can push this onto our dihedral list
    push(@frozen_dihedrals,
       [ $leftside, $atom1, $atom2, $rightside ] );
  }
  }
}
# @frozen_dihedrals now has all possible dihedrals about all of the
# frozen bonds.


# Before we enter the main loop, we have a "free" torsion already
# calculated (which is presumably the lowest energy, since it came from
# an optimization of the whole molecule, and nothing was restricted).
# We can use this value to 'condition' our directory.  Note:  This is not
# strictly true, since we are using a different final single point energy
# for the torsions.  We will take the geometry from the initial optimization,
# and calculate the energy ourselves.
# The contents of our working directory will be:
# Energies <---    A file with a listing of all calculated energies. it
#          will contain one line per calculation, with two space
#          separated fields.  The first field is the angle, the
#          second is the calculated energies.  When there are
#                  a suitable number of acceptable energies,
#                  finish_and_exit() will write them out to angle_vs_energy
# angle_vs_energy < This file is only written out when the program has
#                  determined that all of the Energies are now available.
#
# That's it!  Each time through the loop, we'll check Energies to see if
# there's enough entries to warrant finishing up.  If there is, we simply
# call finish_and_exit().

# The following global variable indicates that a calculation is running.
# Initially, nothing is running, but either the initial calculation (in
# the next block), or a successful pass through the main block will set
# this (forever) to 1.  Early in the main loop, after looking for a message
# file, we check to see if there's a calculation running, if there is, we
# simply restart the loop, otherwise, we proceed through the logic,
# potentially submitting a job that already exists.
my($calculation_running) = 0;

my @logfiles = glob "*.log";

# unless ( -e "Energies" ) {
if ( scalar(@logfiles) == 0) {
  # We assume that if it exists, it already has an entry for the
  # initial file.  This will only fail if somehow the program crashed
  # between opening and writing the string to the file.  This was in
  # the old design.  What the driver does, in practice, is recover
  # very poorly.  Instead, we'll get a list of all of the log files,
  # if there is at least one log file, we'll use the energies from
  # both the initial_optimization and the existing log files to
  # generate the energies files.  If there are enough energies, we
  # regenerate angle_vs_energy, and exit as well.

  my($work_string) = get_last_dihedral_and_energy(
                  "../Initial_optimization.log",
                        @central_dihedral);

  unless(defined($work_string)) {
  die "Call to get_last_dihedral_and_energy() failed";
  }
  my($next_dihedral, undef) = split(/ /, $work_string);

  print "In initial section, next dihedral is $next_dihedral\n";

  # We'll also round $next_dihedral, so we have all 'nice' angles in our
  # energies file.
  $next_dihedral = floor($next_dihedral + 0.5);

  print "Submitting original geometry (dihedral angle is $next_dihedral)\n";

  # And submit this geometry.
  submit_torsion_job($next_dihedral, @central_dihedral, @frozen_dihedrals,
         "../Initial_optimization.log");

  # Finally, we indicate that we have a calculation running by setting
  # our variable.
  $calculation_running = 1;
} else {
  # Here's where we re-build that Energies, and potentially, the
  # angle_vs_energy files.  The first step is to simply delete the
  # file.  Note that this section will fail quite ungracefully if
  # there are unfinished .log files in the directory (or at least I
  # think it will.

  unlink "Energies";  # We don't die if we fail, because if we fail,
                      # the most likely reason is that the file was
                      # already deleted.

  # Add the angles and energies of all of the log files.
  foreach (sort @logfiles) {
  get_angle_energy($_, @central_dihedral);
  }

  # Now, we need only check the size of the Energies file to determine
  # if we can finish_and_exit;
  print "Rebuilding Energies files in initialization\n";

  open TMP, "<Energies" or print "Cannot open Energies for reading, " .
  "even though we just created it, this is an odd logical error" and
  die "Cannot open Energies for reading, but we just created it, " .
      "this is an odd logical error";
  my @tmp = <TMP>;
  close TMP;

  if (scalar(@tmp) >= $sampling_rate) {
  # We need to provide an angle vs energy hash for finish_and_exit;
  my %tmp;
  foreach (@tmp) {
    my ($angle, $energy) = split;
    $tmp{$angle} = $energy;
  }
  print "We appear to have enough torsions to finish in initialization, " .
    "calling finish_and_exit\n";
  finish_and_exit( %tmp );
  }
}

# $lastlog is the name of the last logfile we submitted for
# calculation.  We will check for its existance even if we don't have
# a message file for us.
my $lastlog = undef;

# We'll generate a list of all of the .com files, and look for one
# that does not have a .log file.  If we find one, we'll set $lastlog
# to it, otherwise, we'll leave it undefined.
my %comhash;
my @comlist = glob "*.com";
foreach (@comlist) {
  my($name, undef) = split('.', $_);
  $comhash{$name} = 1;
}
my $foundlog = 0;
my @loglist = glob "*.com";
foreach (@loglist) {
  my($name, undef) = split('.', $_);
```

```
  if (exists($comhash{$name})) {
  next;
  } elsif ($foundlog) {
  print "Found multiple unfinished com files, cannot continue\n";
  die "See runlog.txt for details, cannot continue\n";
  }
  $foundlog = 1;
  $lastlog = "$name.log";
}

MAIN_LOOP: while ($loop_forever) {

  $now = localtime;
  print $now;
  print "\n";
  print "Just beginning main loop, about to reset the alarm and sleep\n";


  # Restart the alarm
  alarm($alarm_timeout);

  # Sleep until we're woken
  sleep unless $got_alarm;
  $got_alarm = 0;

  # The first task in the loop is to see if a job is finished for us, and if
  # it is, we need to read the file and add the new information to our
  # Energies
  my $messageisopen = 0;
  if ( open(TMP, "<../../control/message.$$") ) {
  $messageisopen = 1;
  }

  if ( $messageisopen or defined $lastlog && is_finished($lastlog) ) {

  my $jobname;

  if ($messageisopen) {
    $jobname = <TMP>;
    close(TMP);
    chomp($jobname);
  } else {
    $jobname = $lastlog;
  }

  # Now ... $jobname is a .com file, and we need to turn it into a
  # .log file.  We can't count on the ending being .com, but we can
  # count on the output being a .log (this is the convention we've
  # been using).  This isn't strictly true, now that I've added
  # $lastlog.  However, this section simply replaces the .log suffix
  # with a .log suffix, so it's not really a problem.
  @work_list = split(/\./, $jobname);
  pop(@work_list);
  push(@work_list, "log");
  my($logname) = join(".", @work_list);

  print "Found a message file, or found $lastlog that is finished.  " .
          "The job name was $jobname\n";
  print "Additionally, we'll call get_angle_energy with $logname\n";

  # And delete it if we got the info from the message file.
  if ($messageisopen) {
    unlink("../../control/message.$$");
  }

  my($angle, $energy) =
    split(/ /, get_angle_energy($logname, @central_dihedral));

  if (defined($angle)) {
    $angnrg{$angle} = $energy;

    print "The new angle and energy we entered was: $angle $energy " .
        "(which we updated) in the energies file by our call to " .
      "get_angle_energy\n";

  }
  } else {
  # If there's no message for us, we have no finished jobs, we simply
  # restart the loop.  Note that if a job is already running for our
  # input file, we'll end up resubmitting it (on the first pass).  The
  # local submission program (qdb_local_submit.pl in the developer's case)
  # must check new entries on the que to make sure they're not already
  # running elsewhere.

  print "No message file, and nothing to do, goin back to sleep\n";

  if ($calculation_running) {
    next;
  }

  print "Apparently, there was no calculation running, so we will continue.\n"
.
    "on with the main loop\n";
  }

  # And get our Energies (and angles) if we made it past the last test.
  open(TMP, "<Energies") or die "Critical file error";
  @work_list = <TMP>;
  close(TMP);
  chomp(@work_list);

  # Get the current angles and energies.
  %angnrg = ();
  foreach (@work_list) {
  my($ang, $nrg) = split(/ /, $_);
  $angnrg{$ang} = $nrg;
  }

  my($next_dihedral) = choose_next_dihedral(%angnrg);

  # There will be scattered prints about until we get this up and running.
  print "Chose next dihedral to be: $next_dihedral\n";

  unless(defined($next_dihedral)) {
  # If the function returns undef, it means that we have enough
  # dihedrals, we simply exit.
  finish_and_exit(%angnrg);
  }

  # If we already have a log file for this angle (which would be the case
  # if the daemon was killed, and restarted later), we try to add it to
  # the list, and restart the loop
  if ( -e "torsion_${next_dihedral}.log" ) {

  print "For some reason, we already have a log file for $next_dihedral, so \n"
.
    "we're retrieving information from that, and restarting the loop\n" .
    "again\n";

  get_angle_energy("torsion_${next_dihedral}.log", @central_dihedral);

  # This updates our energies file, so we restart the loop, without
  # sleeping.  We also need to leave ourselves a message, so we know
  # we're done with this one.
  open(TMP, ">../../control/message.$$") or
    die "Unable to open ../../control/message.$$ for writing so we " .
    "could leave ourselves a message, exiting";
  print TMP "$working_directory/torsion_${next_dihedral}.com\n";
  close (TMP);

  print "We just called get_angle_energy - for the side effect of updating\n "
.
    "the energies file, and we'll simply restart the loop (note that\n" .
    "we also left ourselves a message\n";

  $got_alarm = 1;
  next;
  }

  print "Submitting torsion job for $next_dihedral in normal logic of main loop
of function.\n";

  submit_torsion_job($next_dihedral, @central_dihedral, @frozen_dihedrals);

  # And ... indicate that we've started a calculation.
  $calculation_running = 1;

  # Our work for this pass has been completed, we now go back to the
  # beginning of the main loop and go back to sleep

  # Since the block is huge, it was intentionally _not_ indented
}

print "Torsion driver received a SIGQUIT, exiting normally.\n";

unlink ("driver_is_running");

#################################################################
#   TTTTTTTrouble!      RRRRRRemove WWWWWhen DDDDDDDone!          #
#################################################################
close(BLAH);

exit;

###################################################################
#       End of program                                            #
###################################################################

# This function must prepare a call to get_last_dihedral_and_energy() in
# the package g98_functions.pl, and return the results.  This function also
# adds the new energy entry into the Energies file if it finds it.
sub get_angle_energy($\@) {
  my($infilename) = shift;
  my(@work_list) = @{$_[0]}; shift;
  my(@dihedral) = @work_list;

  my($work_string) = get_last_dihedral_and_energy(
                        $infilename, @dihedral);
  unless(defined($work_string)) {
  die "Call to get_last_dihedral_and_energy() failed.  It was called" .
      "from withing get_angle_energy, where filename was $infilename " .
    "and the dihedral was $dihedral[0] $dihedral[1] $dihedral[2] " .
    "$dihedral[3]";
  }

  open(TMP, ">>Energies") or die "Unable to append to Energies in " .
  "working directory";
  print TMP "$work_string\n";
  close TMP;

  return $work_string;
}

# This function takes a hash of angle=>energy values, and does a series
# of manipulations to it.  It's job is to choose the next dihedral.
# Also, it needs to notice energies that are too high, and not use
# them in subsequent calculations, but values halfway in between.  This
# algorithm will not find minima in areas besides directly around the
# angles given to it.  See the rest of the function for more algorithmic
# details.
sub choose_next_dihedral(%) {
  my(%angnrg) = @_;
  my(@energies, $i);

  # We want the angles we return to be integral.  Since we have our own
  # copy of the hash, we'll round each key, and use that hash instead.
  my(%work_hash);
  foreach (keys(%angnrg)) {
  $work_hash{norm_round_angle($_)} = $angnrg{$_};
  }
  %angnrg = %work_hash;

  my($max_angle) = floor( 360 / $sampling_rate + 0.5 );
  my(@angles) = sort { $a<=>$b } keys(%angnrg);

  # Find the minimum energy
  @energies = sort { $a<=>$b } values(%angnrg);
```

```
if ( count_valid_energies(@energies) >= $sampling_rate ) {
return undef;
}               # The scan is complete

my($min_nrg) = $energies[0];

# If we only have the base energy, we can simply return that angle
# plus the maximum scan angle.
if ( $#angles == 0 ) {
return norm_round_angle($angles[0] + $max_angle);
}

# Let's clean up the hash, so the energy is a relative energy.
foreach (@angles) {
$angnrg{$_} -= $min_nrg;
}

# We need (want) to sort the list so the largest gap lies between
# the first and last elements of @angles
my($largest_gap) = $angles[0] - $angles[$#angles] + 360;
my($new_base) = $angles[0];
for $i ( 1 .. $#angles ) {
my($this_gap) = $angles[$i] - $angles[$i - 1];
if ( $this_gap > $largest_gap ) {
  # Mark this base.
  $largest_gap = $this_gap;
  $new_base = $angles[$i];
}
}

# And ... rotate the list;
until ( $angles[0] == $new_base ) {
push ( @angles, shift(@angles) );
}

# If either end of the list is not above the cutoff in energy,
# we can return that energy + or - our $max_angle, unless there
# already exists an energy at that angle, in which case, we simply
# request a calculation for the angle between the two extremes.
if ( $angnrg{$angles[$#angles]} < $nrg_cutoff ) {
unless ( exists($angnrg{$angles[$#angles] + $max_angle}) ) {
  return norm_angle( $angles[$#angles] + $max_angle );
} else {
  return norm_round_angle( ($angles[$#angles] + $angles[0]) / 2 );
}
}

if ( $angnrg{$angles[0]} < $nrg_cutoff ) {
unless ( exists( $angnrg{$angles[0] - $max_angle} ) ) {
  return norm_angle($angles[0] - $max_angle);
} else {
  return norm_round_angle( ($angles[$#angles] + $angles[0]) / 2 );
}
}

# If we get this far, both ends of the list are energies that are
# 'too high' to accept, but there may be multiple such entries, so
# we trim the ends of the list until there's only one such entry.

while ( $angnrg{$angles[$#angles - 1]} > $nrg_cutoff ) {
pop(@angles);
}
while ( $angnrg{$angles[1]} > $nrg_cutoff ) {
shift(@angles);
}

# Ok, our list is trimmed.  The next task is to find the angle that
# gets us as close to the 'end' of our torsion as possible (but no
# closer than $min_sample_angle).  We will take the last two points,
# and do a linear interpolation to guess the next dihedral.  It might
# be more accurate to do a polynomial fit to the last three points,
# but because the distance between the dihedrals is highly variable,
# we would only be able to do that if the points were close together.
# Instead of a series of complicated tests, we do our best with
# the linear fit.

if ( norm_angle($angles[$#angles] - $angles[$#angles - 1]) >
   $min_samp_angle ) {
# We can try to find this 'edge'
my($slope, $intercept) =
  get_line_parameters(
            $angles[$#angles - 1] + 360,
            $angnrg{$angles[$#angles - 1]},
            $angles[$#angles] + 360,
            $angnrg{$angles[$#angles]}
           );

# We have the equation for our line, now to find the angle
# at which the energy is max.  (Because of the anticipated shape
# of the potential, this should actually give an energy _less_
# than the maximum, but close).

my($new_angle) =
  norm_round_angle( ( $nrg_cutoff - $intercept ) / $slope - 0.5 );

# Now, only return our new angle if it's larger than our
# $min_samp_angle
if ( abs($angles[$#angles - 1] - $new_angle) >= $min_samp_angle ) {
  return $new_angle;
}
}

# If we couldn't extend the end of the list, let's try the beginning
if ( norm_angle($angles[0] - $angles[1]) >
   $min_samp_angle ) {
# We can try to find this 'edge'
my($slope, $intercept) =
  get_line_parameters(
            $angles[1] + 360,
            $angnrg{$angles[1]},
            $angles[0] + 360,
            $angnrg{$angles[0]}
           );

  # While troubleshooting this function (and after adding the function
  # norm_round_angle() ), I decided to leave the next line with the
  # ceil call.  I believe this is the correct decision, since we wish
  # to err _towards_ the high energy side of this dihedral.
  my($new_angle) = norm_angle( ceil(
                  ( $nrg_cutoff - $intercept ) / $slope - 0.5 ) );

  # Now, only return our new angle if it's larger than our
  # $min_samp_angle
  if ( abs($angles[1] - $new_angle) >= $min_samp_angle ) {
    return $new_angle;
  }
  }

  # Finally, we have both ends identified, all that's left to do is
  # submit the angle between the largest gap.  If we have only a three
  # member list, with one valid energy, and two out of range, simply
  # return that we're done with new angles, this will give the ft fitting
  # at least three points (both terminal angles and energies).

  if ( $#angles == 2 ) {
  return undef;
  }

  $largest_gap = norm_angle(360 + $angles[2] - $angles[1]);
  $new_base = $angles[1];
  for $i ( 3 .. ( $#angles - 1) ) {
  my($this_gap) = norm_angle ( 360 + $angles[$i] - $angles[$i - 1] );
  if ( $this_gap > $largest_gap ) {
    $largest_gap = $this_gap;
    $new_base = $angles[$i - 1];
  }
  }

  if ( $largest_gap / 2 < $min_samp_angle ) {
  return undef;
  }               # We're done

  # Return the angle halfway in between the requested angles, rounding
  # up and down pseudo-randomly, to prevent generating a bias in the
             # generated torsion

  # Perl automatically calls srand with a suitable seed the first time
# rand is called (only true for versions above 5.0.1, which we already
             # required in the beginning)
  my ($choice) = floor( rand(1) + 0.5 );

  # Find the new base again
  for $i ( 1 .. ($#angles - 1) ) {
  if ( $new_base == $angles[$i] ) {
    # And do our return
    my($return_angle) =
    norm_angle( 360 + $angles[$i] + $largest_gap / 2 );
    return $choice ? floor($return_angle) : ceil($return_angle);
  }
  }

  # If we get this far in the function, the logic is hosed, and we
  # simply die.
  die "Reached the end of choose_next_dihedral(), which should not " .
  "be possible.  Exiting";
}

# This tiny and useful function returns any angle (in degrees) into its
# equivalent angle between 0 and 360;
sub norm_angle($) {
  my($angle) = shift;
  while ( $angle < 0   ) {
  $angle += 360;
  }
  while ( $angle >= 360 ) {
  $angle -= 360;
  }
  return $angle;
}

# The small function both normalizes and rounds the angle it receives.
# If it's working correctly, it should always return an integer.
sub norm_round_angle ($) {
  my($angle) = shift;
  $angle = norm_angle($angle);
  $angle = floor($angle + 0.5);
  return $angle;
}

# This function does everything we need to have done when we're finally
# finished with a calculation.
sub finish_and_exit(\%) {
  my(%work_hash) = %{$_[0]}; shift;
  my(%angnrg) = %work_hash;

  # Exit if we've already written this file
  if ( -e "angle_vs_energy" ) {
  unlink("driver_is_running"); exit;
  }

  # All we need to do is normalize the energies, and output the
  # contents of the received hash to angle_vs_energy
  my($min_energy) = undef;
  foreach ( keys(%angnrg)) {
  unless (defined($min_energy)) {
    $min_energy = $angnrg{$_};
  }
  if ( $angnrg{$_} < $min_energy ) {
    $min_energy = $angnrg{$_};
  }
  }

  # We have the minimum energy, normalize the hash, and finally print
  # it out.
  foreach ( keys(%angnrg)) {
  $angnrg{$_} -= $min_energy;
  }
```

```
  open(TMP, ">angle_vs_energy") or
  die "Unable to open angle_vs_energy for writing, exiting\n";

  foreach (sort {$a<=>$b} keys(%angnrg)) {
  print TMP "$_=>$angnrg{$_}\n";
  }
  close(TMP);

  # Before we leave, make certain to tell the folks we're done.
  open(TMP, ">>../../control/message.$callers_pid") or
  die "Unable to open ../../control/message.$callers_pid " .
    "for appending\n";
  flock(TMP, LOCK_EX);
  seek(TMP, 0, 2);
  print TMP "Torsion scan complete\n";
  flock(TMP, LOCK_UN);
  close(TMP);

  kill SIGALRM, $callers_pid;

  unlink ("driver_is_running");
  die "finish_and_exit() completed\n";
}

# The following function is one of the last 'big job' functions.  Its
# responsibility is to take a new torsion angle to calculate, a dihedral
# (to identify the atoms of interest), and a list of bonds to freeze
# during the optimization at this dihedral angle.  It then finds the
# closest job, and request the local functions (g98_functions in the
# development case) for the geometry of the requested file.  It
# modifies the geometry and finally, requests the local functions to
# prepare a valid input file.  Finally, it adds the request to the
# que.  If the (optional) final argument is received, it means that
# the base geometry should be chosen from that structure, not the
# closest structure (which would be the case for the first calculation of
# this type in a given directory)
sub submit_torsion_job($\@\@;$) {
  my($angle) = shift;

  # This function call was munching the third argument.  I didn't
  # realize this before, but once again, references make an appearance
  # in strange ways.  The list is actually a list of references to
  # lists, so making a copy of it actually gives the future references
  # a chance to munch the original values.  We need to make copies
  # of our own for usage in this function.
  my(@dihedral) = @{$_[0]}; shift;
  my(@work_list) = @{$_[0]}; shift;
  my(@frozen_bonds) = ();
  foreach (@work_list) {
  push(@frozen_bonds, [ @{$_} ]);
  }
  my($optarg) = shift;
  my(%dir_list);
  my($work_value);

  opendir(TMP_DIR, ".") or
  die "Unable to open working directory for a listing";
  while ( $work_value = readdir(TMP_DIR) ) {
  $dir_list{$work_value} = 1;
  }
  closedir(TMP_DIR);

  # Remove the irrelevant entries
  delete($dir_list{"."});
  delete($dir_list{".."});
  delete($dir_list{"driver_is_running"});
  delete($dir_list{"Energies"});

  # Ok, it's here we finally need to settle onto a format for our
  # output files.  Quite arbitrarily, I've decided on:
  # torsion_<angle>.log   ---   Note that as an initial oversight
  # in the development, I've somewhat counted on the output files being
  # .log files, but I'm pretty sure I've been neutral about the other
  # filenames I've used.  When developing functions for other ab
  # initio programs, simply have the functions create a link to the
  # real output file, with a .log extension instead of the default
  # extension for your own ab initio program.

  my(%available_angles);
  foreach (keys(%dir_list)) {
  if ( $_ =~ m/torsion_(\d+)\.log/ ) {
    $available_angles{$1} = $_;
  }
  }

  # And add the (ubiqitous) initial entry.
  # We no longer need to do this, since we re-calculated the original
  # geometry.

  # We now have a list of available angles (and therefore, structures)
  # to choose from ... choose the closest.

  my($min_diff) = 181;
  my($best_angle) = undef;
  foreach (keys(%available_angles)) {

  my($this_diff) = norm_angle($_ - $angle);

  if ( $this_diff > 180 ) {
    $this_diff = 360 - $this_diff;
  }
  if ( $this_diff < $min_diff ) {
    $min_diff = $this_diff;
    $best_angle = $_;
  }
  }

  if ($optarg) {
  # Override the previous logic.
  $available_angles{"override"} = $optarg;
  $best_angle = "override";
  }

  my(@structure) =
  get_optimized_structure($available_angles{$best_angle});

  # Uncomment the next line if you need connectivity for your own
  # implementation (to determine how to rotate around the central dihedral
  # bond, for example).
  # my(%connectivity) =
  #  get_optimized_connectivity($available_angles{$best_angle});

  # If we had to modify the geometry ourselves, this section would
  # get a bit hairy, but we don't need to do that, we can simply
  # submit a request for this calculation, with modified coordinates
  # ( frozen dihedrals and the dihedral of interest ).  In the
  # developer's environment, gaussian 98 does this itself.  If your
  # ab initio program has no similar option, you need to have the
  # local formatting function do the geometry manipulation.  Remember
  # to not rotate portions of molecules if your dihedral is in a
  # ring, but to simply rotate both ends by 1/2 of the requested angle.

  # The function is to be called as:
  # make_dihed_inp_file(<($)new_file_base_name>,<($)new_angle>,
  #                      <(\@)dihedral_of_interest>, <(\@)fragment geometry>,
  #            <(\@)frozen_bonds>,<$rcfilename>)

  my($infilename) =
  make_dihed_inp_file("torsion_$angle", $angle, @dihedral,
           @structure, @frozen_bonds, $rcfilename);

  # And add our full path to infilename
  $infilename = $working_directory . "/" . $infilename;

  # Lastly, we add this to the que
  open(QUE, ">>../../control/que") or
  die "Unable to open the que for appending";

  my($now) = localtime;
  print "$now\n";
  print "About to enter (into the que):
$this_user,$this_host,$$,$infilename\n";

  flock(QUE, LOCK_EX);
  seek(QUE, 0, 2);
  print QUE "$this_user,$this_host,$$,$infilename\n";
  flock(QUE, LOCK_UN);
  close(QUE);

  print "Entry made.\n";

  # Before we leave, set $lastlog to the file we just entered.
  $lastlog = $infilename;

  return;
}


# This function simply returns the number of energies under the cutoff
# for the list provided.
sub count_valid_energies(@) {
  my(@energies) = @_;

  # Sort the list descending
  @energies = sort {$b<=>$a} @energies;

  my($max_energy) = $energies[$#energies] + $nrg_cutoff;

  # Remove invalid energies
  while ($energies[0] > $max_energy) {
  shift(@energies);
  }

  return scalar(@energies);

}

# The following function simply returns m and b from the familiar
# y = mx + b equation.  The four arguments are x1, y1, x2, and y2, in
# that order.
sub get_line_parameters($$$$) {
  my($x1, $y1, $x2, $y2) = @_;

  if ( $x1 == $x2 ) {
  return (undef, undef);
  }

  my($slope) = ( $y2 - $y1 ) / ( $x2 - $x1 );
  my($intercept) = $y1 - $x1 * $slope ;

  return ($slope, $intercept);
}

# This function simply prints out the connectivity, it's useful for
# troubleshooting, but may not be useful in the final program.  Note
# That it prints with the 1 based offsets, to be a bit easier to read
sub print_connectivity(%) {

  my(%con) = @_;

  foreach ( sort {$a<=>$b} ( keys(%con) ) ) {
  my($key) = $_;
  print $key + 1 . "\t---->";
  foreach ( @{$con{$key}} ) {
    print " " . ($_ + 1);
  }
  print "\n";
  }

  return;
}

# The following function prints out all of the information contained
# in a molecule.  It's mainly here for demonstration purposes
sub print_molecule (@) {

  my(@molecule) = @_;
```

```
  print "Begin coordinates\n";
  foreach (@molecule) {
  print $_->[0];
  print ",";
  print join(",", @{$_->[1]} ) . "\n";
  }

  print "Begin connectivity\n";
  print join("\n", @{$molecule[0][2]}) . "\n";

  print "Begin qcodes\n";
  foreach (@molecule) {
  print join("\n", @{$_->[3]} ) . "\n";
  }

  my($needs_sdescrip) = 0;
  my($i) = 0;
  my(@work_list) = ();
  my(%work_hash);
  foreach (@molecule) {
  if ( $_->[4] ) {
    $needs_sdescrip = 1; $work_hash{$i} = $_->[4];
  }
  $i++;
  }
  if ( $needs_sdescrip ) {
  @work_list = sort { $a <=> $b } keys(%work_hash);
  print "Begin stereochemical descriptors\n";
  foreach (@work_list) {
    print "$_ $work_hash{$_}\n";
  }
  }
  return;
}

# The following functions handle signals

# This handler lets the program know if it got an alarm
sub wake_up {
  $got_alarm = 1;
  return;
}

# The following handler ignores SIGHUP, which shells send upon exit.  It
# appears to have the side effect of setting off the alarm.
sub handle_hup {
  return;
}

# This handler allows the program to exit gracefully, though it's not
# likely to be used often, since it is a daemon, after all
sub quit_now {
  $loop_forever = 0;
  $got_alarm = 0;
  return;
}
```

qdb_query_server.pl

```
#!/usr/bin/perl -w

# REMOVE LATER!!  Note that taint checking has been removed for
# debugging purposes.

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

package main;

# I discovered (quite by accident) that this server, though running well in
# all other respects, does not deal well with having directories deleted
# out from under it.  This is a minor problem, and can be overcome by
# having it simply delete entries from it's running hash that are not
# currently available when it refreshes.  Alternately, whenever someone
# removes a directory from the database, they could restart the server,
# though this requires that you remember to do so.  Since (when the
# system is completely up and running) there should _never_ be entries
# removed from the database, this is considered a low priority problem.

# From here on, I'll be using this pragma for all new programs.  It
# requires sensible scoping for all variables.  This means all
# variables should be defined with my().  Local variables aren't good
# enough.  (Unless you need a local variable) If we really need a
# variable that will be global between files, we can declare it after
# temporarily turning off the strict pragma, or fully qualifying it.
use strict;

# (Note:  The first version of this program used files and signals to
# communicate.  I believe I've removed all remnants of that implementation,
# but if something looks odd, it's likely a dinosaur from that time)

# To begin with, this program was copied from v1.2 of qdb_calculate.pl,
# since is will be structurally similar.  It is designed to be a database
# query server for the qdb.  The first attempt to search the database
# became slow very quickly, as the entire database on the drives was
# searched for each query.  It was decided that it would be much more
# efficient to simply read the contents of the database _only_ when
# they changed, and to keep the relevant indexes in memory the rest of
# the time.  This implementation will not be fully scaleable either,
# but should be very useful while the number of entries in the
# database is < 10000 (when the memory usage of the daemon would be
# approximately 165MB).  Obviously, above that size, the implementation
# could be reworked, probably by some form of hashing the relevant
# query information.  Just a note on the size.  In order to actually do
# the ab initio calculations for 10,000 entries in the database, 20
# computers like jubjub (in the authors home cluster) would need to be
# caluculating 24/7 for about 8 years.  The memory usage may not ever
# be a problem.

# NOTE:  This is more or less an update from the previous section.  Using
# files and signals turned out to be quite a bit slower than was acceptable
# (with a c client, and this server, it was taking about 0.75 seconds per
# query ... way too slow).  As a result, we will be using internet
# sockets for this program.  With a bare perl client and server passing
# short strings back and forth, we were able to get 2800 transactions per
# second with the programs running on separate machines on the same LAN.
# This is plenty fast, and that model will be inserted into this server
# code.  The handler for SIGALRM and SIGQUIT will be left in place, and
# the program will still write it's host and id in the control directory
# of $main::db_path.  The meaning for SIGALRM will be slightly different,
# however.  Following is an excerpt from the scaffold server program:

my($alarm_timeout) = 600;  # How frequently the alarm (to re-index the
# database) goes off.
my($server_timeout) = 10; # How long the server waits for a connection
# before checking if an alarm went off
my($client_timeout) = 60 ;   # How long the client socket waits before
# closing the socket on the remote client

# Some comments on the above variables: For the production scale query
# server, the client_timeout should be set for a (reasonably) short
# time.  The server could be expanded to allow the client to request a
# different timeout than the default, but this isn't really necessary
# for now.  The reason it should be set for a reasonably short time is
# that the server is unable to take any connections while it's dealing
# with a single client.  This will only become a limitation when the
# server must be extremely busy, and the client does not terminate the
# connection properly.  Another solution to this problem involves
# having the server spawn children to handle the requests.  This
# modification would be nice to do once we atually have many users.
# For development, there will only be one user at a time, so this is
# also considered a low priority.

# The $server_timeout should be set to a reasonably short time, i.e., 60
# seconds.  The reason is that it really determines how often the reindexing
# timer is checked.  In order to actually do a reindex, it's necessary for
# the server to timeout.  Note that if the server times out, and the alarm
# hasn't been checked, all that happens is that the alarm is checked, and
# the socket goes back to a listening state.

# Finally, the alarm_timeout can be set for a reasonably long time (600-6000
# seconds).  This number is approximately how long the system goes before
# updating the database.  If I understand the structure correctly (I think
# I do), then the maximum time between updates will be $server_timeout +
# $alarm_timeout.  This is not _strictly_ true.  The server will never
# reindex as long as the time between client socket connections is less than
# $server_timeout.  This is the main reason to have this at a relatively
# short time.

# The reason the program was more or less copied from qdb_calculate is that
# it, also, is a daemon, and designed to run constantly while the system
# is in use.  A basic (revised, to reflect the changed mentioned in the
# above section) outline of the program follows:
# 1)  Start up, initialize the relevant search fields, register its PID
#     in the file $db_path/control/qdb_query_server.pid.  Also, add an
#     entry (appended) to the file $db_path/control/qdb_query_server.log.
#     Note that several environment variables will need to be read from
#     .qdb_checkrc
# 2)  Initialize an internet socket to listen to.  We are using a port
#     number that is currently unregistered with IANA, see comments near
#     the declaration of $PORT for details.  After registering the socket,
#     the program listens to that socket until it either gets a connection,
#     or the server socket times out.
# 3)  Each time it's accept call returns, it does the following.  First,
#     it checks to see if it's actually had a client connect to it.  If
#     it has, it will server the client, as outlined in the next section
#     labelled Syntax:
#     If there is no client waiting, it checks to see if it's received
#     a SIGALRM (The handler sets a global variable).  If it has, it
#     reindexes the database as necessary.

# Syntax:
#     The format of the queries are contained on single lines, and are
#     as follows:
#     <some query string> [{list1}] [{list2}]
# In more detail:
# <get> [# of matches] <atom|bond> <match> {qcode_1_list} [{qcode_2_list}]
# or:
# <get> <charge> <charge_type> <directory> <atom_number>

# As functionality is added to the server, the new language options
# should also be provided above.

# Where the angle bracketed items are manditory, and the square
# bracketed item is optional.  If omitted, it will find a single
# best match, and if there is a tie, it will pick the best match.  If
# there are identically good matches, it returns at least as many
# matches as are requested.  Note that the curly braces are required
# in the query to delimit the qcodes.
# Finally, the server sends back a tab separated list, where each item
# contains the directory the match was found in, the line number
# on which it was found, and the qcode deviance, as .  For bond
# matches, it returns 2 atoms.  (And every pair of atoms with the
```

```
# same level of matching).

eval { require 5.6.1 }
  or die <<MESSAGE;
####################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
####################################################################
MESSAGE

# Included libraries
require "/home/radke/dev/ff/general/rc_file_handling.pl";

# For some reason, the next line stopped working??
# use lib "../shlib/cpmodule";

# The following gets us ready to use the CFUNCS package, which is simply
# our own interface to c functions that should not be duplicated.
# use lib qw(/home/radke/dev/ff/shlib/cpmodule/);
use lib qw(/home/radke/dev/ff/shlib/cpmodule);
use CFUNCS;

# Global variables (here for reference and my declarations).  Many are
# simply declared with my at their first use.  Note that it would be
# 'better' to not declare all of the following variables my() in the
# outside level of the package.  Unfortunately, this program was written
# originally without the use strict pragma, and the variables were all
# completely global.  Someday this could be cleaned up, but should be
# considered low priority.  If the server does not understand the
# command passed by the client, it should (in all cases now, hopefully)
# simply close the socket quietly.

# Note:  The following are global, as functions need to use them.
# As development continues, I have to apologize (mostly to myself)
# for the growing number of global variables.  Basically, as I keep
# expanding the program, it makes more and more sense to move some
# of my already working functions into their own subroutines.
# Someday, this could ceratainly be cleaned up.

$main::db_path = "";
$main::qdb_index_last_access = ();
$main::qdb_index_qcodes = ();
$main::loop_forever = 1;
$main::got_alarm = 1;  # Set to one so we're re-indexed the first time
# through, this variable is changed when the
# alarm handler catches an alarm.
$main::client = 0;;
@main::queried_atom1_qcode = ();
@main::queried_atom2_qcode = ();

# After many many sections of debugging, I've decided that for large
# programs (such as this one) defining a global debug variable, for
# optional inclusion (printing out extra information)
$main::debug = 1;

my($tolerance) = 0.000000000001;
my($my_pid) = $$;
my(@work_list);
my($work_value);
my($atom1);
my($atom2);
my(%qdb_index_connectivity) = ();
my($i);
my(%work_hash);
my($line);
my($directory);
my($this_directory);
my($command);
my(@this_query);

# Function Prototypes
sub functions::output_bond_matches (@); # This line likely does
                                        # nothing, since all functions
                                        # have been moved to main.
                                        # Regardless, it will be left
                                        # in place until it can be
                                        # removed, and the
                                        # functionality tested (low
                                        # priority)

# We need to eliminate the possibility of malicious clients entering
# very long queries in order to run up the memory on the machine, the
# following variables are declared for that portion of the program.

my($max_input_size) = 16 * 1024;
# 16K, this should be long enough.  This can be set to arbitrarily large
# values, but the larger it is, the more memory (and processor time) will
# be used up before the server decides that the query is too long and
# (possibly) malicious.

# For explicitly requested long queries, allow a larger input string
my($long_query_max_size) = 1024 * 1024;

my($bytes_read) = 0;

# Just in cast the hostname returns the entire hostename, we split off the
# first part (users are more likely to have registered the names with ssh
# in this case
@work_list = split(/\./, $ENV{"HOSTNAME"}, 0);
my($this_host) = $work_list[0];

# Process the .qdb_checkrc file
open("RCFILE", '<.qdb_checkrc') or die
  "Unable to open .qdb_checkrc ... exiting\n";

# Get the port number from the .qdb_checkrc file
my($PORT) = &read_scalar("RCFILE", "query_server_port");

$main::db_path = &read_scalar("RCFILE", "db_path");
defined($main::db_path) or die
```

```
  "Unable to find db_path in .qdb_checkrc file ... exiting\n";

# Since we want to run a secure server, we need to 'launder' the $db_path
# so we can use the -T (does taint checks) option when starting the server.
# This is not a recommended way of getting the program to compile, but I
# believe that every use of $db_path is secure.  It is possible that someone
# could change the .qdb_checkrc file, in which case it may be possible to
# implement malicious file writes.  I am (intentionally) not handling this
# security breach
$main::db_path =~ m%[\w/\.]+%;  # %'s used since / is in the pattern.
$work_value = $&;
$main::db_path = $work_value;

# Note also that the port number is considered tainted now as well, as
# long as the port number is only numbers, and the value is between
# 1025 and 49151 (inclusively).  The only security breach I perceive here
# is that a malicious user that changes the .qdb_checkrc file could make
# the server run on any port.  Since the server itself is only capable of
# doing malicious file writes (to qdb_query_server.log), the only other
# potential problem might be blocking some legitimate program from using
# a port.
$PORT =~ m/[\d]+/;
$PORT = $&;
unless ($PORT >= 1025 and $PORT <= 49252) {
  die "Invalid port $PORT read from .qdb_checkrc file, exiting\n";
}

close("RCFILE");


###########################################################################
#                  Begin normal part of program                           #
###########################################################################

# Note startup in log.
append_to_log("qdb_query_server starting up");
# Fix log permissions just in case
chmod(0664, "$main::db_path/control/qdb_query_server.log");

# Register our host and pid.
open(TEMP, ">$main::db_path/control/qdb_query_server.pid");
print TEMP "$my_pid\n";
close(TEMP);
chmod(0664, "$main::db_path/control/qdb_query_server.pid");
open(TEMP, ">$main::db_path/control/qdb_query_server.host");
print TEMP "$this_host\n";
close(TEMP);
chmod(0664, "$main::db_path/control/qdb_query_server.host");

# Start here, initializing the server.  The following includes
# initialization information for the internet socket server.

# For the purpose of this client and server, we will be using port 5561,
# which is currently unassigned by the iana (Internet Assigned Numbers
# Authority, www.iana.org).  This will of course be included as a #define
# in C program headers, and a variable in the perl programs.  When the
# software is likely to become public, we should get the iana to reserve
# this port for our own use.
# The specific address is: http://www.iana.org/assignments/port-numbers

# This value is now read from the .qdb_checkrc file
# my($PORT) = 5561;

use IO::Socket;
use Net::hostent;

# Note:  There was a problem with the perl installation on the DEC cluster
# the author uses for some development.  The last line of this comment
# caused several import errors.  As a result, the value of $CRLF is set
# manually.  If I understand correctly, this should give identical
# results to the 'proper' way.
# Update:  The author has installed perl 5.6.1 locally, and this is
# no longer a problem.
use Socket qw(:DEFAULT :crlf);
# use Socket qw(:DEFAULT);

# Set the input record separator to the internet line terminator.  Note
# that its client's responsibility to provide this separator, otherwise,
# Their request will be read until the receiver portion of the server
# times out, or it receives more than $max_input_size of data.  This
# design allows badly written clients to 'hang' the server for as long
# as the timeout allows.  Shorten the $client_timeout for a quick patch
# to this problem.

# Save this value for future usage, sometimes, we need to change it
# to the 'normal' separator, so we can read files.
my($oldsep) = $/;
$/ = $CRLF;

# The following section was written to allow the server to run on almost
# any host that has a $HOSTNAME in it's environment.  In the case of my
# (Home) computer, some additional code was necessary in order to get
# the machine's IP address into the $server constructor (In my case,
# My machines $HOSTNAME doesn't exist on local DNS's, and needed to
# be retrieved from the /etc/hosts file.  It could almost
# certainly be re-written, but since it only runs once at startup, it
# shouldn't create a performance issue.

my($hostname) = $ENV{'HOSTNAME'};

# The following manual taint check taken from:
# http://www.perldoc.com/perl5.6/pod/perlsec.html
# This expression 'launders' $hostname, so it can be used to create the
# socket
if ($hostname =~ /^([-\@\w.]+)$/) {
  $hostname = $1;
} else {
  append_to_log("Bad hostname: \"$hostname\" retrieved from " .
        "environment.  Refusing to start server");
  die "Bad hostname, exiting";
}

# I had problems with the /etc/hosts lookup portion of this server
# during development.  That's my best guess at the problem, I'm really
```

```
# not positive what the problem was, except that I couldn't use $hostname
# in LocalAddr => "$text_ip_addr:$PORT" for some reason (see a couple of
# lines down).  (Sheesh, that inet_ntoa(inet_aton()) call looks horrific!)
my($text_ip_addr) = inet_ntoa(inet_aton($hostname));

my($server) = IO::Socket::INET->new( Proto => "tcp",
                   LocalAddr => "$text_ip_addr:$PORT", Listen => SOMAXCONN,
Reuse => 1);
# End new(er) section


if (!$server) {
  append_to_log("Unable to open port for listening, exiting");
  die "Critical error:  Unable to open port for listening in server.pl";
}
$server->timeout($server_timeout);
# Server doesn't need to autoflush, its client will do that.

# Uncomment the following for troublshooting
print "Server now accepting up to " . SOMAXCONN .
  " clients on port " . $server->sockport() . "\n";
append_to_log ("Server now accepting up to " . SOMAXCONN .
         " clients on port " . $server->sockport());

# Initialize the qcode hash, and index the qcodes before we start listening
reindex_qcodes();

# Reset the alarm, and start it
$main::got_alarm = 0;
alarm($alarm_timeout);

# Begin forever loop
SERVER: while($main::loop_forever) {

  # Note that we are using the object oriented versions of the perl
  # internet socket implementation.  It's quite easier to use, and
  # not significantly slower (though I've not explicitly tested
  # this hypothesis)
  $main::client = $server->accept();

  if (!defined($main::client)) {
  # See if we've timed out, re-index if we have, and reset
  # everything else.
  if ($main::got_alarm) {
    reindex_qcodes();
    # Reset the alarm, and restart it
    $main::got_alarm = 0;
    alarm($alarm_timeout);
  }
  } else {
  # The client is connected and alive, do everything this program
  # is basically designed to do.

  # Set the client to autoflush (old perl versions don't set this
  # automatically.
  $main::client->autoflush(1);

  # The client socket inherits the timeout from the server, so
  # we reset it to the client timeout value.
  $main::client->timeout($client_timeout);

  my($do_loop) = 1;

  # The basic flow of the server is that it waits for a command,
  # processes it, and returns a response it will do this until
  # the client side times out, or it receives a "close" command

  my($hostinfo) = gethostbyaddr($main::client->peeraddr);

  append_to_log("Connection received from " . $hostinfo->name );

  # Uncomment the following for debugging
  print "Connection received from " . $hostinfo->name . "\n";

  CLIENT:  while ($do_loop) {

    # Clean up the variables before we start the loop
    undef($work_value);
    undef(@this_query);

    # Only accept the first $max_input_size characters, if the
    # total number was entered, ignore the query.

    # Note:  this description is obsolete!  See the next block.
    # There is no simple way to refuse to read lines over a
    # certain amount.  As a result, I'll be reading in
    # 'blocks', with the flag set to MSG_PEEK, and searching
    # the blocks for the "$CRLF" sequence.  If the sequence is
    # not found, the block is read with the flag set to 0, and
    # the loop continues until the total size read is above the
    # max.  If the $CRLF _is_ found in the block, that last
    # block is read with the normal filehandle (<>), and
    # concatenated to the building string.  This sequence
    # prevents strings that are too long from being received
    # by the server, and leaves the buffer in place for the
    # next read.

    # The previous section was written before I learned the
    # magic of the select() call.  Note that there is a warning
    # (that I don't fully understand) that can be read at:
    # http://www.perldoc.com/perl5.6/pod/func/select.html.
    # Regardless, the design is similar to what was read above,
    # except that the socket _is_ read from, unless we see a
    # $CRLF in our MSG_PEEK step.  This resets the socket for
    # the next select() call, and keeps the reads relatively
    # small.  Note that the $client_timeout is used in the
    # select() call, but it's unlikely that it would even have
    # time to elapse, unless the client intentionally 'stays
    # on the line'.  Even so, we'll see to it that the connection
    # is still hung up after that time.  For more information
    # on this use of select, see perlfunc::select.

    # Note on mixing buffered input with select().  Eventually,
    # I ran into a problem that was most likely a result of the
# warning mentioned in the previous paragraph.  As it turned
# out, the <> operator was eating more than up to and including
# the initial $CRLF characters.  That small portion has been
# re-written to explicitly read until just before the $CRLF,
# and record it, then read length($CRLF) characters, and
# discard them.

$command = $work_value = '';
$bytes_read = 0;
my($read_side) = '';
my($nfound);
my($leave_now) = 0;

while ( $bytes_read < $max_input_size and
    $work_value !~ /$CRLF/g and !$leave_now and
    defined(fileno($main::client)) ) {
vec($read_side, fileno($main::client), 1) = 1;
$nfound = select($read_side, undef, undef, $client_timeout);

if ($nfound) {
  $main::client->recv($work_value,
          $max_input_size, MSG_PEEK);
  if (length($work_value) == 0) {
  # The remote client has closed its connection
  # on us.  We will do the same so we can take
  # future requests.
  $command = $work_value = "close";
  $leave_now = 1;
  } elsif ($work_value !~ m/$CRLF/ ) {
  $main::client->recv($work_value, length($work_value));
  $command .= $work_value;
  $bytes_read = length($command);
  } else {
  # Leave, since we have something to read now,
  # (which includes the CRLF at the end we've
  # been looking for
  $leave_now = 1;
  }
} else {
  # The client has stayed on the line, but refuses to hang
  # up (after $client_timeout) has elapsed.  Forcefully
  # close its connection.
  $command = $work_value = "close";
  $leave_now = 1;
}
}
if ( $bytes_read >= $max_input_size ) {
# We were sent a command far longer than any should
# be, so we'll close the socket quietly
print "Maximum input record size exceeded, ignoring this"
  . " query\n";
append_to_log("Maximum sized input received from " .
       $hostinfo->name . ", this is most likely " .
       "malicious, terminating connection");
$command = $work_value = "close";
} elsif ( $command eq "close" ) {
# This may have been received, or it may have been
# set from reading a 0 length after return from the
# select call.  Either way, ignore it, and let it be
# processed normally.
} else {
# There are no circumstances that would prevent further
# processing of this command, so we simply get it ready
# for further processing.  If the socket is still open,
# read the final $CRLF section from it, if not, just
# get ready for the rest of the processing.  Ok, this is
# (probably) where the warnings about using select
# come from.  The <> operator is _not_ just reading
# up to the $CRLF, which means it's our job to take
# just that many characters off of the socket, and leave
# the rest intact.

# Reset the position in work_value, this is a fresh run
pos($work_value) = 0;
if (defined(fileno($main::client) ) and
  $work_value =~ m/$CRLF/g ) {
  my($another_string);
  $main::client->recv($another_string,
          pos($work_value) - length($CRLF) );

  # Append it to the command
  $command .= $another_string;

  # And discard the $CRLF from the input stream.
  $main::client->recv($another_string, length($CRLF) );
} else {
  append_to_log( "Unknown error condition encountered " .
        "while getting input from the client, " .
        "resetting connection\n");
  close $main::client;
  last CLIENT;
}
$work_value = $command;
}

# Clean up the string for processing
if (defined($work_value)) {
# The following pattern simply deletes non-legal
# characters.
$work_value =~ s/[^[:alnum:] .\-}{]//g;
$command = $work_value;
# Since the spaces around the {'s and }'s are optional,
# we need to explicitly make them members of the list
@this_query = split(/([}{])/, $command);
$work_value = join(" ", @this_query);
@this_query = split(/ +/, $work_value);
# And strip extraneous spaces
$command = join(" ", @this_query);
}

# Note:  It's no longer necessary to keep a record of the
# last query, since the TCP/IP protocol guarantees the
# data arrives in order.
```

```
if ($main::debug) {
# print "Processing (cleaned up) command string\n";
# print "$command\n";
# print "This query has elements:\n";
# print join("\n", @this_query);
}

if (!defined($this_query[0]) or $this_query[0] =~ /^close/ ) {
# Close the socket
$do_loop = 0;
close $main::client;
last CLIENT;
} elsif ($command =~ /^get [0-9]*[ ]*atom match/ ) {
# Match atoms here.

if ($main::debug) {
  print "In atom matching section\n";
}

# If there was a number of matches requested, initialize
# it.
my($requested_matches) = 1;  #default value
shift(@this_query);
$work_value = shift(@this_query);
if ( $work_value =~ /^[\d]+$/ ) {
  $requested_matches = $work_value;
  if ($requested_matches == 0) {
  $requested_matches = 1;
  }
  # And shift off another value
  shift(@this_query);
}

# And one more shift to get rid of the "match" member
shift(@this_query);

# Now, this is a bit tricky, but we need to extract the
# qcode from what we received.  Since the {'s and }'s are
# actually list elements, it really isn't so tricky.  We
# simply need to do our error checking to make sure we
# have decimal numbers between the braces.

# If the next item in our list is not '{', we got a bad
# input.
$work_value = shift(@this_query);
if ($work_value ne "{") {
  append_to_log("Bad input received from client (no " .
        "opening \"{\" in qcode specification, " .
        "ignoring this query and hanging " .
        "up");
  last CLIENT;
}
# Now, all of the fields up until the next (and final )
# "}" should be numbers to push onto the requested
# qcodes list.

my(@queried_atom_qcodes) = ();
while ( defined($work_value = shift(@this_query) )
    and $work_value ne "}" ) {

  if ( $work_value !~ /^-?[\d]+[\.]?[\d]*$/ ) {
  append_to_log("Non number qcode encountered in " .
          "atom matching, ignoring query" .
          " and hanging up");
  last CLIENT;
  }

  push(@queried_atom_qcodes, $work_value);
}

unless (defined($work_value) and $work_value eq "}") {
  append_to_log("No trailing \"}\" found when reading " .
        "qcode, ignoring this query and " .
        "hanging up");
  last CLIENT;
}

# Ok, our entire command line is processed, and any further
# data in the command line is discarded.  Begin matching

my($match_position) = 0;

my(@list_of_matches) = ();
my(@last_list_of_matches) = ();
my($extra_matches_needed);

while (($this_directory, $work_value) =
     each(%main::qdb_index_qcodes) ) {

  for ($i = 0;
     $i <= $#{$main::qdb_index_qcodes{$this_directory}};
     $i++) {

  # Here's the test
  if (abs($main::qdb_index_qcodes{$this_directory}
      [$i][$match_position] -
      $queried_atom_qcodes[$match_position])
    < $tolerance) {
    push(@list_of_matches, $this_directory .
       " " . $i);
  }
  }
}
# The initial 'guess' list is built.

@last_list_of_matches = @list_of_matches;
my(@working_list_of_matches) = ();
$match_position++;

# Note that in the following loop control, there is no
# check to see if we've grown larger than the qcodes
# recorded in %qdb_index_qcodes.  This is acceptable,
# as the uninitialized values will return 0, which it
# a number that qcodes cannot converge towards (I believe),
# and even if they did, eventually, we would run out of
# size on th $queried_atom_qcodes.  From the Camel, 3rd
# edition, page 7:  "If you use a variable that has never
# been assigned a value, the uninitialized variable
# automatically springs into existence as needed.
# Following the principle of least surprise, the
# variable is created with a null value, either "" or 0.
# Depending on where you use them, variables will be
# interpreted automatically as strings, as numbers or
# as "true" and "false" values ..."
until ($#list_of_matches < $requested_matches
     or $match_position > $#queried_atom_qcodes) {
  # Pare down the list by requiring more and more
  # places in the qcode to match

  foreach $work_value (@list_of_matches) {
  my($line_number);
  ($directory, $line_number) =
    split(/ /, $work_value, 2);
  if ($match_position <=
    $#{$main::qdb_index_qcodes
       {
      $directory}[$line_number]}
    and abs($main::qdb_index_qcodes{$directory}
        [$line_number][$match_position] -
        $queried_atom_qcodes[$match_position])
    < $tolerance) {
    push(@working_list_of_matches,
       "$directory $line_number");
  }
  }
  # The new (hopefully smaller) list is created, get ready
  # for the next iteration
  @last_list_of_matches = @list_of_matches;
  @list_of_matches = @working_list_of_matches;
  @working_list_of_matches = ();
  $match_position++;
}
$match_position--;
# This guarantees that the match position is still the
# index to the last known match in the list_of_matches
# In all cases, @list_of_matches will need the deviance
# appended to it, if we need to do this for the
# @last list of matches, we'll do it there.
for ($i = 0; $i <= $#list_of_matches; $i++) {
  ($directory, my($atom)) =
  split(/ /, $list_of_matches[$i], 3);
  $list_of_matches[$i] .=
  " " .
  CFUNCS::get_qcode_deviance(
    $main::qdb_index_qcodes{$directory}[$atom],
              \@queried_atom_qcodes);
}
if ($#last_list_of_matches == -1) {
  # This means the last_list of matches is empty,
  # indicating that there were no matches to the
  # query, simply return a blank line to the
  # client
  print $main::client "$CRLF";
} elsif ( ($#list_of_matches) + 1 >= $requested_matches ) {
  # This is the simplest case.  If there were too many
  # matches all the way up, simply output them all.  The
  # client may choose to ignore additional matches.
  print $main::client join("\t", @list_of_matches) .
  "$CRLF";
} else {
  # The only case left is that the current list of matches
  # has fewer matches than requested, and the last list
  # of matches has more than requested.  In this case, we
  # need to remove any matches from the last list that
  # are in the first list, then search the next qcode for
  # the minimum deviation.  Those with minimum deviations
  # will be added to the .out file (after the currentl
  # list of matches).

  # Put last_list_of_matches into a hash, for faster
  # searching.
  %work_hash = ();
  foreach $work_value (@last_list_of_matches) {
  $work_hash{$work_value} = 1;
  }
  # Now remove ones mentioned in @list_of_matches
  foreach $work_value (@list_of_matches) {
  delete($work_hash{$work_value});
  }

  # Now, assign the value of each key to the
  # deviation as defined in get_qcode_deviance()

  @work_list = keys(%work_hash);
  foreach $work_value (@work_list) {
  ($directory, my($atom)) =
    split(/ /, $work_value, 2);
  $work_hash{$work_value} =
    CFUNCS::get_qcode_deviance (
      $main::qdb_index_qcodes{$directory}[$atom],
              \@queried_atom_qcodes );
  }

  # We now have everything we need to create the output
  # file.  First, we print the best list, then, we add
  # all values from lowest to biggest until we've
  # reached the requested number of matches.  Note that
  # we will print all values that have equal deviation,
  # even if it gives more output than requested.

  # Reverse the hash, so it can be sorted on deviation.
  # Note that each entry will be a reference to a list,
  # since there is not a one to one correspondence
  # of deviation to directory.
  my(%work_hash2) = ();

  while ( ($directory, $work_value) =
      each(%work_hash) ) {
```

```
  push(@{$work_hash2{$work_value}}, $directory);
  }

  # Finally, we can get to the business of outputting
  # the values we've gotten.
  $extra_matches_needed = $requested_matches -
  $#list_of_matches - 1;

  if ($#list_of_matches != -1) {
  print $main::client join("\t", @list_of_matches);
  }

  @work_list = sort {$a <=> $b} keys(%work_hash2);
  for ( $i = 0; $i <= $#work_list &&
    $extra_matches_needed > 0; $i++ ) {
  if ($i != 0 or $#list_of_matches != -1) {
    print $main::client "\t";
  }
  foreach (@{$work_hash2{$work_list[$i]}}) {
    print $main::client "$_ $work_list[$i]\t";
  }
  $extra_matches_needed -=
    ($#{$work_hash2{$work_list[$i]}} + 1)
  }
  # And finally, the closing newline;
  print $main::client "$CRLF";
}

# End atom matching section

} elsif ($command =~ /^get [\d]*[ ]*bond match/ ) {
# Match bonds here

if ($main::debug) {
  print "Matching bonds\n";
}
# This was yet another horrificly difficult error to find.
# I failed to re-initialize the queried_atom_qcodes before
# each iteration, and we simply appended new requests to
# the existing qcode.  They are initialized here.

# Using a program really help to find the odd errors.
# 6-18-2002, the bond matching section was returning the
# matches in an arbitrary order, instead of in the order they
# were requested.  Bonds are almost never symmetric, so this
# caused real problems in implementation.
@main::queried_atom1_qcode = ();
@main::queried_atom2_qcode = ();

# If there was a number of matches requested, initialize
# it.
my($requested_matches) = 1;  #default value
shift(@this_query);
$work_value = shift(@this_query);
if ( $work_value =~ /^[\d]+$/ ) {
  $requested_matches = $work_value;
  # And shift off another value
  shift(@this_query);
}

# And one more shift to get rid of the "match" member
shift(@this_query);

# See the comments in the atom match section for details
# on extracting the qcode from the line received.

# If the next item in our list is not '{', we got a bad
# input.
$work_value = shift(@this_query);
if (!defined($work_value) or $work_value ne "{") {
  append_to_log("Bad input received from client (no " .
        "opening \"{\" in qcode specification," .
        " ignoring this query and hanging " .
        "up");
  last CLIENT;
}
# Now, all of the fields up until the next (and final )
# "}" should be numbers to push onto the requested
# qcodes list.

while ( defined($work_value = shift(@this_query) )
    and $work_value ne "}" ) {

  if ( $work_value !~ /^-?[\d]+[\.]?[\d]*$/ ) {
  append_to_log("Non number qcode encountered in " .
          "atom matching, ignoring query");
  last CLIENT;
  }
  push(@main::queried_atom1_qcode, $work_value);
}

unless (defined($work_value) and $work_value eq "}") {
  append_to_log("No trailing \"}\" found when reading " .
        "qcode, ignoring this query and " .
        "hanging up");
  last CLIENT;
}

# And basically do the whole thing again, but with the
# second queried qcode.

# If the next item in our list is not '{', we got a bad
# input.
$work_value = shift(@this_query);
unless(defined($work_value) and $work_value eq "{") {
  append_to_log("No leading \"{\" found when reading " .
        "qcode, ignoring this query and " .
        " hanging up");
  last CLIENT;
}

# Now, all of the fields up until the next (and final )
# "}" should be numbers to push onto the requested
# qcodes list.

while ( defined($work_value = shift(@this_query) )
    and $work_value ne "}" ) {

  if ( $work_value !~ /^-?[\d]+[\.]?[\d]*$/ ) {
  append_to_log("Non number qcode encountered in " .
        "atom matching, ignoring query");
  last CLIENT;
  }

  push(@main::queried_atom2_qcode, $work_value);
}

unless(defined($work_value)) {
  append_to_log("No trailing \"}\" found when reading " .
       "qcode, ignoring this query and " .
       " hanging up");
  last CLIENT;
}

# Ok, our entire command line is processed, and any further
# data in the command line is discarded.  Begin matching

if ($main::debug) {
  print "After initialization:\n";
  print "Qcode1 = " . join(" ", @main::queried_atom1_qcode) . "\n";
  print "Qcode2 = " . join(" ", @main::queried_atom2_qcode) . "\n";
}

# Initialize working variables for the next section
my($match_position) = 0;
my(@list_of_matches) = ();
my(@last_list_of_matches) = ();
my(@working_list_of_matches) = ();

if ($main::debug) {
  print "After initialization, working list of matches has ";
  print scalar(@working_list_of_matches);
  print " members (0 expected)\n";

  print "Before building initial guesses, list_of_matches is " .
  "(expect nothing):\n";
  foreach (@list_of_matches) {
  print @{$_}->[0] . "\t" . join(" ", @{@{$_}->[1]}) . " ... " .
    join(" ", @{@{$_}->[2]}) . "\t";
  print "\t";
  print "Match at:$match_position\n";
  }
}

# Begin building the initial list of possible matches.

while (($this_directory, $work_value) =
     each(%main::qdb_index_qcodes)) {
  my(%qcode_1_match) = ();
  my(%qcode_2_match) = ();

  for ($i = 0;
     $i <= $#{$main::qdb_index_qcodes{$this_directory}};
     $i++) {
  if (abs($main::qdb_index_qcodes{$this_directory}
      [$i][$match_position] -
      $main::queried_atom1_qcode[$match_position]) <
    $tolerance) {
    $qcode_1_match{$i} = 1;
  }
  if (abs($main::qdb_index_qcodes{$this_directory}
      [$i][$match_position] -
      $main::queried_atom2_qcode[$match_position]) <
    $tolerance) {
    $qcode_2_match{$i} = 1;
  }
  }

  if ($main::debug) {
  print "In $this_directory (original guesses): " .
        "qcode_[1|2]_match = " .
      scalar(keys(%qcode_1_match)) . ", " .
      scalar(keys(%qcode_2_match)) .
      " (members)\n";
  }

  # We've found all of the matches, but not verified
  # the connectivity yet.  The next steps are:
  # 1) Remove duplicates in %qcode_2_match (relative
  #    to %qcode_1_match)
  # NOTE:  This just isn't this simple.  If the
  #        very first qcode is the same, this removes
  #        _all_ of the qcode2_matches, which is
  #        definately not what is intended.
  # CONCLUSION: The section of code that removed
  #             duplicates has been commented out. It
  #             is clear that the same atom will never
  #             be connected to itself (in
  #             Connectivity.raw) so this should not be
  #             a problem
  # 2) In each iteration, push the values onto the
  #    list of possible matches.  Then loop through
  #    these possible matches.
  # 3) Check the associative array
  #    %qdb_index_connectivity for the presense of
  #    an entry for this directory
  # 4) If found, check the date, and update if
  #    necessary.
  # 5) If not found, add an entry to that hash
  # 6) In any case, for each pair of values in the
  #    qcode_#_match hashes, determine if they're
  #    connected.  If they are connected, add the
  #    values to the @list_of_matches
  # 7) Initial list of matches list is done!

  # @work_list = keys(%qcode_1_match);
  # foreach $work_value (@work_list) {
  # if (exists($qcode_2_match{$work_value} ) ) {
```

```
  # delete($qcode_2_match{$work_value});
  # }
  # }

  # @list_of_matches has the following structure:
  # @list_of_matches is a list of arrays.
  # $list_of_matches[0] (or any other legal value)
  #     is a reference to an array.
  # $list_of_matches[0][0] is the name of the
  #     directory for which the other information is
  #     valid.
  # $list_of_matches[0][1] is a reference to the list
  #     matches to the first qcode in this directory
  # @{$list_of_matches[0][1]} is the actual list of
  #     matches to qcode one in the current directory

  # Note:  Only add this directory to the list if
  # there are some matches!
  if (%qcode_1_match and %qcode_2_match) {
  push(@list_of_matches,
     [  $this_directory,
      [sort {$a <=> $b} keys(%qcode_1_match)],
      [sort {$a <=> $b} keys(%qcode_2_match)]
     ]);
  }
}

my(@match_1_list);
my(@match_2_list);
my(@work_list_2);

if ($main::debug) {
  print "After finding all directories with matches, " .
      "list_of_matches is:\n";
  foreach (@list_of_matches) {
  print @{$_}->[0] . "\t" . join(" ", @{@{$_}->[1]}) .
        " ... " . join(" ", @{@{$_}->[2]}) . "\t";
  print "\t";
  print "Match at:$match_position\n";
  }
}

# Now we work through the new list, and make sure we
# have a current index of the connectivity for each
# directory in which we need is
foreach (@list_of_matches) {
  my($mod_time);
  $directory = $_->[0];
  # Note we cannot use $_[0], as the underscore
  # interferes with this common shorthand.
  @match_1_list = @{$_->[1]};
  @match_2_list = @{$_->[2]};

  if (exists($qdb_index_connectivity{$directory})) {
  # Check the date and update as necessary
  $mod_time = time -
    (-M "$main::db_path/$directory/Connectivity.raw")
    * 86400;
  if ($mod_time >
    $qdb_index_connectivity{$directory}[0][2]) {
    # Re-read the relevant information
    open(TEMP, "<$main::db_path/$directory/" .
       "Connectivity.raw") or
       append_to_log("Warning, could not open " .
              "$this_directory/Connectivity.raw for " .
              "reading, daemon terminating") and
            die "Could not open a critical file, " .
              "check log for details";
    # Because we're doing file access here, we
    # need to be reading the normal separators.
    $/ = $oldsep;

    @work_list = <TEMP>;

    $/ = $CRLF;
    close(TEMP);

    for ($i = 0; $i <= $#work_list; $i++) {
    @work_list_2 = split(/ /,
              $work_list[$i]);
    $qdb_index_connectivity{$directory}
      [$i][0] = $work_list_2[0];
    $qdb_index_connectivity{$directory}
      [$i][1] = $work_list_2[1];
    }
    # And ... insert the time marker
    $qdb_index_connectivity{$directory}[0][2]
    = time;
  }
  } else {
  # Create this entry
  open(TEMP, "<$main::db_path/$directory/" .
     "Connectivity.raw") or
       append_to_log("Warning, could not open " .
            "$this_directory/Connectivity.raw for " .
            "reading, daemon terminating") and
          die "Could not open a critical file, " .
            "check log for details";

  # Change to 'normal' separator.
  $/ = $oldsep;
  # And read, then change back.
  @work_list = <TEMP>;
  $/ = $CRLF;

  close(TEMP);
  for ($i = 0; $i <= $#work_list; $i++) {
    @work_list_2 = split(/ /, $work_list[$i]);
    $qdb_index_connectivity{$directory}[$i][0] =
    $work_list_2[0];
    $qdb_index_connectivity{$directory}[$i][1] =
    $work_list_2[1];
  }
  # And ... insert the time marker
  $qdb_index_connectivity{$directory}[0][2] = time;
  }

  # Now, compare the lists of matches to see if any
  # of them are connected.  If they are, record them
  # for further searching.

  # Note:  In the next section, it would have been
  # (a bit) more efficient to store the connectivity
  # in a hash (for searching), but the syntax (a hash
  # within a hash) could have gotten nightmareish,
  # so I've opted for the list handling approach.

  # Since the match lists will be searched repeatedly,
  # place them into hashes temporarily.
  my(%work_hash_1) = ();
  my(%work_hash_2) = ();
  foreach (@match_1_list) {
  $work_hash_1{$_} = 1;
  }
  foreach (@match_2_list) {
  $work_hash_2{$_} = 1;
  }
  for ($i = 0;
     $i <= $#{$qdb_index_connectivity{$directory}};
     $i ++) {

  # Long if statements - it basically says if the
  # atoms are connected for any two memebers of
  # the hash, do the true bit (add to a list of
  # matches)

  if ( (exists($work_hash_1
         {
          $qdb_index_connectivity{$directory}[$i][0]})
      and exists($work_hash_2
           {
            $qdb_index_connectivity{$directory}[$i][1]}))
     or
     ( exists($work_hash_2
          {
           $qdb_index_connectivity{$directory}[$i][0]})
       and exists($work_hash_1
           {
            $qdb_index_connectivity{$directory}[$i][1]}))) {
    push(@working_list_of_matches,
       [$directory,
      $qdb_index_connectivity{$directory}
      [$i][0],
      $qdb_index_connectivity{$directory}
      [$i][1]
       ]
      );
  }
  }
}

# For a single test case, the connectivity and list of matches
# seems to be acceptable at least up until this point.
# Another example will also need to be taken through manually
# and verified.

if ($main::debug) {
  # Begin here, trying to make sure the original guesses are decent
  print "Printing Original Working list of Matches\n";
  foreach (@working_list_of_matches) {
  my(@geomave);
  $geomave[0] = sqrt(
    CFUNCS::get_qcode_deviance(
      $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
      \@main::queried_atom1_qcode) *
          CFUNCS::get_qcode_deviance(
            $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
            \@main::queried_atom2_qcode) );
  $geomave[1] = sqrt(
    CFUNCS::get_qcode_deviance(
      $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
      \@main::queried_atom2_qcode) *
    CFUNCS::get_qcode_deviance(
      $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
      \@main::queried_atom1_qcode) );
  print @{$_}->[0] . "\t" . @{$_}->[1] . "\t" . @{$_}->[2] . "\t";
  print $geomave[0] > $geomave[1] ? $geomave[0] : $geomave[1];
  print "\t";
  print "Match at:$match_position\n";
  }
}

# The initial guess list is completely built.  Now we
# simply need to go through, increasing the number of
# matches until we get the right number, or bracketed
# the number of requested matches.  This section should
# look pretty similar to the atom matching bit.
@last_list_of_matches = @list_of_matches =
  @working_list_of_matches;

@working_list_of_matches = ();
$match_position++;    # Start matching next member of
# qcode vector

until ($#list_of_matches < $requested_matches
     or $match_position > $#main::queried_atom1_qcode
     or $match_position > $#main::queried_atom2_qcode) {

  # Pare down the list by requiring more and more
  # places in the qcodes to match

  foreach (@list_of_matches) {
  ($directory, $atom1, $atom2) = @{$_};
  if ($main::debug) {
    print "Shrinking list-directory:$directory\tatom1:" .
    "$atom1\tatom2:$atom2\n";
  }
```

```
  # Big if statement coming up.  It requires that
  # There is an exact match at $match_position
  # for _both_ atoms.  This section needed to be
  # added to.  We need to not only check that there
  # is a match at both positions, be we need to
  # check 'both ways' (note that the section after
  # the or is the same as the first, with atom1
  # and atom2 permuted).
  if (abs($main::qdb_index_qcodes{$directory}
      [$atom1][$match_position] -
      $main::queried_atom1_qcode[$match_position])
    < $tolerance
    &&        # Higher Precedence
    abs($main::qdb_index_qcodes{$directory}
      [$atom2][$match_position] -
      $main::queried_atom2_qcode[$match_position])
    < $tolerance
    or        # Lower precedence
    abs($main::qdb_index_qcodes{$directory}
      [$atom2][$match_position] -
      $main::queried_atom1_qcode[$match_position])
    < $tolerance
    &&        # Higher Precedence
    abs($main::qdb_index_qcodes{$directory}
      [$atom1][$match_position] -
      $main::queried_atom2_qcode[$match_position])
    < $tolerance ) {
    # We have a match at the next level,
    # push it onto @working_list_of_matches.
    # Be careful to push the _values_ onto the
    # list, and not a reference to the vales
    # in the previous list.
    push(@working_list_of_matches, [$directory,
                $atom1, $atom2]);
  }
  }

  @last_list_of_matches = @list_of_matches;
  @list_of_matches = @working_list_of_matches;
  @working_list_of_matches = ();
  if ($main::debug) {
  print "Match position:$match_position\t list:";
  print 1 + $#list_of_matches;
  print "\tlast list:";
  print 1 + $#last_list_of_matches;
  print "\n";
  }
  $match_position++;
}

# The two lists should be initialized now

if ($main::debug) {
  print "Printing list of Matches\n";
  foreach (@list_of_matches) {
  my(@geomave);
  $geomave[0] = sqrt(
      CFUNCS::get_qcode_deviance(
      $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
          \@main::queried_atom1_qcode) *
        CFUNCS::get_qcode_deviance(
    $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
          \@main::queried_atom2_qcode) );
  $geomave[1] = sqrt(
        CFUNCS::get_qcode_deviance(
          $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
          \@main::queried_atom2_qcode) *
        CFUNCS::get_qcode_deviance(
          $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
          \@main::queried_atom1_qcode) );
  print @{$_}->[0] . "\t" . @{$_}->[1] . "\t" . @{$_}->[2] . "\t";
  print $geomave[0] > $geomave[1] ? $geomave[0] : $geomave[1];
  print "\t";
  print "Match at:$match_position\n";
  }

  print "Printing last list of Matches\n";
  foreach (@last_list_of_matches) {
  my(@geomave);
  $geomave[0] = sqrt(
        CFUNCS::get_qcode_deviance(
          $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
          \@main::queried_atom1_qcode) *
        CFUNCS::get_qcode_deviance(
          $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
          \@main::queried_atom2_qcode) );
  $geomave[1] = sqrt(
        CFUNCS::get_qcode_deviance(
          $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
          \@main::queried_atom2_qcode) *
        CFUNCS::get_qcode_deviance(
          $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
          \@main::queried_atom1_qcode) );
  print @{$_}->[0] . "\t" . @{$_}->[1] . "\t" . @{$_}->[2] . "\t";
  print $geomave[0] > $geomave[1] ? $geomave[0] : $geomave[1];
  print "\t";
  print "Match at:$match_position\n";
  }
  # We look pretty good up until here.
}

$match_position--;
# This guarantees that the match position is still the
# index to the last known match

# Variables used in the next section
my(%work_hash);
my($extra_matches_needed);

if ($#last_list_of_matches == -1) {
  # This means the last_list_of_matches is empty,
  # indicating that there were no matches to the
  # query, simply return a blank line to the
  # client
  print $main::client "$CRLF";
} elsif ($#list_of_matches + 1 >= $requested_matches) {
  # Simplest case.  Output all of the matches.
  # The client may ignore additional matches if it
  # so desires.
  output_bond_matches(@list_of_matches);
  print $main::client "$CRLF";
} else {
  # The only case left is that @list_of_matches has
  # fewer matches than requested, and
  # @last_list_of_matches has too many.  We need to
  # write out all of the @list_of_matches, and the
  # most closely fitting @last_list_of_matches.

  if ($main::debug) {
  print "In output section:  Checking values\n";
  print "Printing list of matches\n";
  foreach (@list_of_matches) {
    my(@geomave);
    $geomave[0] = sqrt(
          CFUNCS::get_qcode_deviance(
      $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
      \@main::queried_atom1_qcode) *
          CFUNCS::get_qcode_deviance(
            $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
            \@main::queried_atom2_qcode) );
    $geomave[1] = sqrt(
          CFUNCS::get_qcode_deviance(
            $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
            \@main::queried_atom2_qcode) *
          CFUNCS::get_qcode_deviance(
            $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
            \@main::queried_atom1_qcode) );
    print @{$_}->[0] . "\t" . @{$_}->[1] . "\t" . @{$_}->[2] . "\t";
    print $geomave[0] > $geomave[1] ? $geomave[0] : $geomave[1];
    print "\t";
    print "Match at:$match_position\n";
  }

  print "Printing last list of Matches\n";
  foreach (@last_list_of_matches) {
    my(@geomave);
    $geomave[0] = sqrt(
          CFUNCS::get_qcode_deviance(
            $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
            \@main::queried_atom1_qcode) *
          CFUNCS::get_qcode_deviance(
            $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
            \@main::queried_atom2_qcode) );
    $geomave[1] = sqrt(
          CFUNCS::get_qcode_deviance(
            $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[1]],
              \@main::queried_atom2_qcode) *
          CFUNCS::get_qcode_deviance(
            $main::qdb_index_qcodes{@{$_}->[0]}[@{$_}->[2]],
            \@main::queried_atom1_qcode) );
    print @{$_}->[0] . "\t" . @{$_}->[1] . "\t" . @{$_}->[2] . "\t";
    print $geomave[0] > $geomave[1] ? $geomave[0] : $geomave[1];
    print "\t";
    print "Match at:$match_position\n";
  }
  }

  # Put @last_list_of_matches into a hash for fast
  # handling.  Once again, references come back to
  # bite us.  We need to make the hash use text, and
  # not values for keys, since the address of two
  # copies of the same values is bound to be different.
  # Note that we'll enter the deviance as we create the
  # hash, for later reversing.
  %work_hash = ();
  foreach (@last_list_of_matches) {
  my(@geomave);
  $geomave[0] =
    sqrt(CFUNCS::get_qcode_deviance
       ($main::qdb_index_qcodes
      {
       @{$_}->[0]}[@{$_}->[1]],
      \@main::queried_atom1_qcode) *
       CFUNCS::get_qcode_deviance
       ($main::qdb_index_qcodes
      {
       @{$_}->[0]}[@{$_}->[2]],
      \@main::queried_atom2_qcode)
      );
  $geomave[1] =
    sqrt(CFUNCS::get_qcode_deviance
       ($main::qdb_index_qcodes
      {
       @{$_}->[0]}[@{$_}->[1]],
      \@main::queried_atom2_qcode) *
       CFUNCS::get_qcode_deviance
       ($main::qdb_index_qcodes
      {
       @{$_}->[0]}[@{$_}->[2]],
      \@main::queried_atom1_qcode)
      );
  $work_hash{join(" ", @{$_})} =
    $geomave[0] > $geomave[1] ?
    $geomave[0] :
    $geomave[1] ;
  }

  # Remove values duplicated in @list_of_matches
  foreach (@list_of_matches) {
  delete($work_hash{join(" ", @{$_})});
  }

  # The following section will be (once again)
  # rewritten.  Instead of using some RMS deviation,
  # it will use get_qcode_tolerance() to determine
  # what the 'best' match is.  This was written before
  # the deviance was immediately assigned (see prevous
  # section)
```

```
  # The values of work_hash are now basically how
  # 'good' the match is.  (If you didn't catch the
  # horrendous expression, it's the geometric mean
  # of the match).  The higher the value, the
  # better the match.

  # See atom matching section for more details, this
  # implementation basically mimics it.  The basic idea
  # is that we reverse the hash, stacking entries that
  # have identical 'deviations'.
  my(%work_hash2) = ();

  while (($directory, $work_value) =
     each(%work_hash) ) {
  push(@{$work_hash2{$work_value}},
     [ split(/ /, $directory)] );
  }

  if ($#list_of_matches == -1) {
  $extra_matches_needed = $requested_matches;
  } else {
  $extra_matches_needed = $requested_matches -
    $#list_of_matches;
  }
  if ($main::debug) {
  print "Outputting list of matches:\n";
  }
  output_bond_matches(@list_of_matches);
  if ($main::debug) {
  print "Done outputting list of matches:\n";
  print "Requested $requested_matches matches, need " .
    "$extra_matches_needed more\n";
  }

  @work_list = sort {$b <=> $a} keys(%work_hash2);
  if ($main::debug) {
  print "Worklist of keys is:" . join("x", @work_list) . "\n";
  }
  for ( $i = 0; $i <= $#work_list &&
    $extra_matches_needed > 0; $i++ ) {
  output_bond_matches(@{$work_hash2{$work_list[$i]}});
  $extra_matches_needed -=
    ($#{$work_hash2{$work_list[$i]}} + 1)
  }

  # And ... the closing newline;
  print $main::client "$CRLF";
}

# End bond matching section

} elsif ($command =~
     /^get charge ([\w]+)[ ]+([\w]+-[\d]+)[ ]+([\d]+)/ ) {
# Get charge here.

# If there was a number of matches requested, initialize
# it.
my($charge);
my($charge_type) = $1;
my($charge_dir) = $2;
my($atom_number) = $3;

my($fname) = "$main::db_path/$charge_dir/charges.$charge_type";

unless ( -r $fname and -T $fname ) {
  append_to_log("File $fname is either not available, or is not " .
        "an ascii text file, closing connection");
  close $main::client;

  if ($main::debug) {
  print "Returning prematurely (closing the connection) due to " .
    "the file $fname not being available\n";
  }

  last CLIENT;
}

# Here, we also need to check the qcodes of all the atoms in
# the given directory.  It is possible that the charge for the
# atom requested needs to be symmetrized before we actually
# return the charge.

my (@queried_atom_qcode) =
  @{$main::qdb_index_qcodes{$charge_dir}->[$atom_number]};

# Now, scan through all of the qcodes in this directory, and
# record each atom number that has a match.
my(@matching_atoms);

TESTLOOP:  for( $i = 0; $i <= $#{$main::qdb_index_qcodes{$charge_dir}};
        $i++ ) {
  my(@this_qcode) = @{$main::qdb_index_qcodes{$charge_dir}->[$i]};
  my($j);

  for ($j = 0; $j <= $#queried_atom_qcode and
     $j <= $#this_qcode; $j++) {

  if ( $queried_atom_qcode[$j] != $this_qcode[$j] ) {
    next TESTLOOP;
  }
  }

  push(@matching_atoms, $i);
}

# If at this point, we have no matching atoms, there's a
# serious logical error in the previous section of code.
# Since we took the qcode from one of the atoms specifically,
# there should definately be at least one match.
if (scalar(@matching_atoms) == 0) {
  append_to_log("Critical logical error in charge matching section")
  and die "Critical logical error, see log for details";
}

open(CHARGES, "<$fname") or
  append_to_log("Critical file error:  Unable to open $fname in " .
        "charge match section") and
          die "Critical file error, see log for details";

my ($line, $i, $othersep);
$i = 0;
$othersep = $/;
$/ = "\n";

my(@charge_file_contents) = <CHARGES>;
chomp(@charge_file_contents);
close(CHARGES);
$/ = $othersep;

# Now, we just get the average charge and return it.
my $sum = 0;
foreach (@matching_atoms) {
  $sum += $charge_file_contents[$_];
}

$charge = $sum / scalar(@matching_atoms);

if ($atom_number > $#charge_file_contents ) {
  append_to_log("Atom number received by remote client " .
        "($atom_number) was too large for the molecule " .
        "in the directory, closing connection");
  close $main::client;

  if ($main::client) {
  print "Returning prematurely (closing connection) due to " .
    "an atom number too large for the file\n";
  }

  last CLIENT;
}

print $main::client "$charge$CRLF";

if ($main::debug) {
  # print "Critical values:\n";
  # print "\tcharge      = $charge\n";
  # print "\tcharge_type = $charge_type\n";
  # print "\t$charge_dir = $charge_dir\n";
  # print "\tatom_number = $atom_number\n";
  # print "\tfname       = $fname\n";

  print "Completed charge match\n";
}

# End charge matching section

} elsif ($command =~ /^get long query ([\d]*)/ ) {

if ($main::debug) {
  print "In get long query\n";
}

# Note that $message_length really represents the
# _maximum_ that will be received.  It still needs
# to be checked to see that it's reasonable, however.
# We actually don't need to check this value, as the
# server will simply refuse to accept anything larger
# than the maximum size.
my($message_length) = $1;

$command = $work_value = '';
$bytes_read = 0;
my($read_side) = '';
my($nfound);
my($leave_now) = 0;

# Ok, I've had quite a bit of trouble getting this next
# section to work, and it seems to be a race type of
# conditon.  The only thing I'm doing different than
# the part that works (see the beginning of the main
# loop) is that the client is expected to send two
# queries in a row.  Sometimes the select call that
# follows (which is identical to the one in the beginning)
# hangs until timeout, even though there's information
# in the socket to be read.  This may be what the
# warnings about not using buffered I/O with select were
# about, but despite the warnings, I'll 'patch' the problem
# by simply doing an initial select call with a very
# short timeout --- I don't think this will work either.

# This problem was solved.  It seems that using the <>
# operator earlier was munching more than it was supposed
# to (most likely related to the warning about using
# select with buffered I/O).  We use unadorned, unbuffered,
# and simple reads now, and seem to have no more problems.

while ( $bytes_read < $long_query_max_size and
    $work_value !~ /$CRLF/g and !$leave_now and
    defined(fileno($main::client)) ) {
  vec($read_side, fileno($main::client), 1) = 1;
  $nfound = select($read_side, undef, undef,
          $client_timeout);

  if ($nfound) {
  $main::client->recv($work_value,
           $long_query_max_size, MSG_PEEK);

  if (length($work_value) == 0) {
    # The remote client has closed its connection
    # on us.  We will do the same so we can take
    # future requests.
    $work_value = $command = "close";
    $leave_now = 1;
  } elsif ($work_value !~ /$CRLF/ ) {
    $main::client->recv($work_value,
             length($work_value) );
    $command .= $work_value;
```

```
      $bytes_read = length($command);
    } else {
      # Leave now, since we have a line end to
      # be reading ... of with you!
      $leave_now = 1;
    }
    } else {
    # The client has stayed on the line, but refuses
    # to hang up (after $client_timeout) has elapsed.
    # Forcefully close its connection.
    $leave_now = 1;
    $work_value = $command = "close";
    }
  }
  if ( $bytes_read >= $long_query_max_size ) {
    # We were sent a command far longer than any should
    # be, so we'll close the socket quietly
    print "Requested message length exceeded, ignoring this"
    . " query\n";
    append_to_log("Maximum sized input received from " .
          $hostinfo->name . ", this is most likely " .
          "malicious, terminating connection");
  } elsif ( $command eq "close" ) {
    # This may have been received, or it may have been
    # set from reading a 0 length after return from the
    # select call.  Close the socket forcefully, and
    # start all over.
    close $main::client;
    last CLIENT;
  } else {
    # There are no circumstances that would prevent further
    # processing of this command, so we simply get it ready
    # for further processing.  If the socket is still open,
    # read the final $CRLF section from it, if not, just
    # get ready for the rest of the processing.  Ok, this is
    # (probably) where the warnings about using select
    # come from.  The <> operator is _not_ just reading
    # up to the $CRLF, which means it's our job to take
    # just that many characters off of the socket, and leave
    # the rest intact.

    # Reset the position in work_value, this is a fresh run
    pos($work_value) = 0;
    if (defined(fileno($main::client) ) and
      $work_value =~ m/$CRLF/g ) {
    my($another_string);
    $main::client->recv($another_string,
             pos($work_value) - length($CRLF) );

    # Append it to the command
    $command .= $another_string;

    # And discard the $CRLF from the input stream.
    $main::client->recv($another_string, length($CRLF) );
    } else {
    append_to_log("Unknown error condition " .
           "encountered while getting input " .
           "from the client, " .
           "resetting connection.\n");
    close $main::client;
    last CLIENT;
    die "Unknown error condition encountered while " .
      "getting input from my client\n";
    }
  }

  # Now, we have the command, and simply need to get it
  # ready to finish processing.

  # Put spaces around the brackets
  $command =~ s/{/ { /g;
  $command =~ s/}/ } /g;

  # And ... pick the mode and get going
  if ($command =~ /^asymmetry/) {
    # Process asymmetry request

    if ($main::debug) {
    print "Processing asymmetry request, command is $command\n";
    }

    # distance_from_center and requested qcodes are kept
    # in parallel lists for readability of handling.
    my(@requested_qcodes);
    my(@distance_from_center);
    my(%potential_matches) = ();
    my(%returned_atoms) = ();
    my(%s_descriptors) = ();

    my(@this_query) = split(/ +/, $command);
    shift(@this_query);  # And throw 'asymmetry' away.
    my($match_directory) = shift(@this_query);
    my($central_atom) = shift(@this_query);
    my($tolerance) = shift(@this_query);

    # And ... get the rest of the values from the
    # command.  Most of the time, there should only be
    # one, but we're handling multiple cases for more
    # complicated queries.
    while (scalar(@this_query)) {

    push(@distance_from_center, shift(@this_query));

    if ( shift(@this_query) ne "{" ) {
      append_to_log("No leading { to mark " .
          "beginning of qcode in asymmetry match, " .
          "closing socket");
      close $main::client;
      last CLIENT;
    }
    # Start reading qcodes
    my(@work_list) = ();
    my($work_value);
    while ( ($work_value = shift(@this_query)) ne "}" &&
      scalar(@this_query) ) {
    push(@work_list, $work_value);
  }

  # Make sure there was a trailing brace
  if ( $work_value ne "}" ) {
    append_to_log("No leading } to mark " .
         "end of qcode in asymmetry match, " .
         "closing socket");
    close $main::client;
    last CLIENT;
  }
  # And push this list onto requested_qcodes
  push(@requested_qcodes, [@work_list] );

  }

  # We need to guard against a malicious or careless
  # client telling us that the asymmetric atom is on
  # the other side of the universe
  # ($distance_from_center[$i] is huge), as this will
  # result in the server working very hard for a long
  # time, mainly counting.  If any of those distances
  # are greater than the number of atoms in the directory,
  # we quietly close our connection

  for (my($i) = 0; $i <= $#requested_qcodes; $i++) {
  if ( $distance_from_center[$i] >
     scalar( @{ $main::qdb_index_qcodes
          {
           $match_directory} } ) ) {
    append_to_log("Too large of a search range (" .
         $distance_from_center[$i] .
         ") received in asymmetry request, " .
         "closing connection\n");
    close $main::client;
    last CLIENT;

  }
  }

  # We appear to be initialized fine.  Note that testing
  # has been with only a single distance from atom,
  # qcode pair.

  for (my($i) = 0; $i <= $#requested_qcodes; $i++) {
  # This is the workhorse, the response to the
  # client will be in the form:
  # <y <s_descrip>|n> [y <s_descrip>|n] ...
  # With one answer per distance/qcode pair.  Note
  # that as each atom has a yes returned for it,
  # it will be marked as 'not searchable', so
  # several different atoms with the same qcodes
  # can be returned.  The atom marked will be the
  # one with the 'best' match, as determined by
  # CFUNCS::get_qcode_deviance(\@, \@)
  my(@atoms_at_requested_distance) = ();

  # There are certainly more efficient ways to do
  # the following neighbor searches (likely something
  # like a hash of lists, where the list members are
  # references to other hash entries (bonds) ), but
  # the following should be much more clear.

  # The initial implementation used lists for
  # last_pass_atoms and this_pass_atoms, but what
  # we want is only one mention per atom, and using
  # a hash for this makes much more sense.
  my(%last_pass_atoms);
  $last_pass_atoms{$central_atom} = 1;
  my(%this_pass_atoms) = ();
  my(%visited_atoms);
  $visited_atoms{$central_atom} = 1;
  my($this_atom);
  my($batom1);    # For bonded atom
  my($batom2);

  for (my($j) = 1; $j <= $distance_from_center[$i]
    ; $j++) {

    # After having to come back to this section of code
    # after some time away, it was apparent that a better
    # description was necessary.  The lists held in
    # @{$qdb_index_connectivity{<directory>}} are in the form
    # of "<atom1> <atom2>".  The next loop simply looks
    # through all of the bonds to see if we're connected to
    # the current atom in the loop, and not looking at an
    # already visited atom.
    while ( ($this_atom, undef) =
        each(%last_pass_atoms) ) {
    for (@{$qdb_index_connectivity
        {
         $match_directory} } ) {
      $batom1 = @{$_}[0];
      $batom2 = @{$_}[1];
      if ($this_atom == $batom1) {
      unless ($visited_atoms{$batom2}) {
        $this_pass_atoms{$batom2} = 1;
      }
      } elsif ($this_atom == $batom2) {
      unless ($visited_atoms{$batom1}) {
        $this_pass_atoms{$batom1} = 1;
      }
      }
    }
    }
    while ( ($this_atom, undef) =
        each(%this_pass_atoms) ) {
    $visited_atoms{$this_atom} = 1;
    }
    %last_pass_atoms = %this_pass_atoms;
    %this_pass_atoms = ();
  }
```

```
# We need to handle the special case of 0 atoms
# away (it doesn't make much sense, but we'll
# handle it to deal with the strange request,
# "Am I in the database at this position?"
if ($distance_from_center[$i] == 0) {
  $this_pass_atoms{$central_atom} = 1;
}

# Okies, before we get tolerances for everything,
# we need to read in the Stereochemical_descriptors
# file to see if it marks any of the atoms we
# know about as asymmetric.  This may give us
# an opportunity to bow out early.

unless( -e "$main::db_path/$match_directory/" .
    "Stereochemical_descriptors") {
  # We do make other assumptions about it,
  # like it's readable, etc.  These problems
  # should probably eventually be guarded
  # against.
  print $main::client "n ";
  next;
}

open(TEMP, "<$main::db_path/$match_directory/" .
   "Stereochemical_descriptors") or
     append_to_log("Unable to open $match_directory/".
           "Stereochemical_descriptors after it was " .
           "established the file existed.  Server " .
           "will shut down") and
             die "Unable to open critical file, see log " .
             "for details";

# Reset separators for file reading mode:
my($oldsep) = $/;
$/ = "\n";

@work_list = <TEMP>;
close(TEMP);
$/ = $oldsep;

# Hash it
my(%work_hash) = ();
for (@work_list) {
  (my($key), my($value)) = split(/ +/, $_);
  # Grab only the first character of value
  ($value, undef) = split(/ */, $value, 2);
  $work_hash{$key} = $value;
}
%s_descriptors = %work_hash;

%potential_matches = %last_pass_atoms;

for $work_value (keys(%potential_matches)) {
  if (exists($work_hash{$work_value})) {
  $potential_matches{$work_value} =
    $work_hash{$work_value};
  } else {
  delete($potential_matches{$work_value});
  }
}

# Now, we need to get the deviances (as defined by
# CFUNCS::get_qcode_deviance) for the atoms left,
# also --- let's move them to a hash with a 'better'
# name.

while ( ($this_atom, undef) =
    each(%potential_matches) ) {

  $potential_matches{$this_atom} =
  CFUNCS::get_qcode_deviance(
               $requested_qcodes[$i],
               $main::qdb_index_qcodes
               {
              $match_directory}[$this_atom]);
}

# Get rid of any matches that have already been
# returned
while ( ($this_atom, undef) =
    each(%potential_matches) ) {
  if (exists($returned_atoms{$this_atom})) {
  delete($potential_matches{$this_atom});
  }
}

# Finally, reverse the hash, see if the closest
# is in tolerance range, and return an appropriate
# answer.  Also, mark that answer as returned,
# so it it not returned again.  Note, this might
# clobber some matches, but we're only returning
# one of them, and the rest will show up again,
# so this isn't a problem
%work_hash = ();
while ( (my($key), my($value))
    = each(%potential_matches) ) {
  $work_hash{$value} = $key;
}
@work_list = sort {$b <=> $a} keys(%work_hash);

# And finally, do the deed.
# Note:  Yet another lesson to be learned here,
# be careful of getting empty lists!

if (defined($work_list[0]) and
  $work_list[0] >= $tolerance) {
  $returned_atoms{$work_hash{$work_list[0]}} = 1;

  # Remember, if we're going to send a yes, we
  # need to tell the client what the s_descriptor
  # for the requested atom was.
  if ( defined( $s_descriptors
        {
              $work_hash{$work_list[0]}} ) ) {
        print $main::client "y " .
          $s_descriptors{$work_hash{$work_list[0]}} .
          " ";
        } else {
        # s_descriptors wasn't available??  This
        # means that we got a match when there wasn't
        # one, and indicates and the client should
        # just give up.
        append_to_log(
                "Critical logic error.  Exiting.  " .
                "While deciding return value in " .
                "asymmetry match, a test appears " .
                "to disagree with previous tests, " .
                "this indicates a program error, and " .
                "need to be repaired"
               );
        die "Critical program error, see log for " .
          "details";

        }
      } else {
        print $main::client "n ";
      }
      }

      # And finally, we can finish the query.
      print $main::client "$CRLF";

      # End asymmetry extended query section

    } else {
      append_to_log("Unrecognized command:  \"$command\". " .
           "received from " . $hostinfo->name .
           "while processing long query");
      close $main::client;
    }

    } else {
    # We received an invalid command.  We'll silently close
    # the socket, and record the bad entry.  If we wish to
    # in the future, we can block connections from the offending
    # host, or even domain.

    if ($main::debug) {
      print "Received invalid command, closing socket (ran out of " .
           "cases to test for ..., or command not recognized.)\n" .
        "The command was: $command\n";
    }

    append_to_log("Unrecognized command:  \"$command\". " .
           "received from " . $hostinfo->name .
           "while processing base level query");
    close $main::client;
    }
  }
  }
}

append_to_log("Program received a QUIT signal, exiting normally");
print "Program received a QUIT signal, exiting normally\n";

exit 0;

######################################################################
#       End of program                                               #
######################################################################

# I was calling these functions in a different package, but I'll leave
# them in main.  Once again, this is because the program was retro-
# fitted with the use strict pragma
# package functions;

# The following functions are the signal handles for the program.
# Re-entrant functions are not handled thoroughly in Perl, so it it
# important to have as little in the functions as possible, as even
# changing the values of global variables is not re-entrant.  Unfortunately,
# this means that we will not be logging signal receipt, only exiting the
# program entirely, and handling individual queries.  Note that SIGPIPE
# also needs to be handled, since it is raised when the client on the
# other side dies suddenly, and we'd rather not die in that case.

use sigtrap 'handler', \&wake_up, "ALRM";
use sigtrap 'handler', \&handle_hup, "HUP";
use sigtrap 'handler', \&quit_now, "QUIT";
use sigtrap 'handler', \&handle_hup, "PIPE";

# This is the alarm clock handler, it basically does nothing but return
# control to the program.  It does prevent sleeping if the program catches
# the signal while it's actually busy (very rare).
sub wake_up {
  $main::got_alarm = 1;
  return;
}

# The following handler simply ignores the hangup request, though it appears
# to have the side effect of ending the alarm timer as well.
sub handle_hup {
  return;
}

# The following handler allow normal termination of the program
sub quit_now {
  $main::loop_forever = 0;
  $main::got_alarm = 0;
  return;
}

# The following function was more or less copied from qdb_local_submit.pl
# (my_warn), and will be used to enter log entries.  Note that we have
# chosen to _not_ initialize the file to be blank at startup.  This file
# should be cleaned occasionally.

sub append_to_log {
```

```
  my($message) = join("",@_);
  my($right_now);
  # Open our log file.

  open (MY_LOG, ">>$main::db_path/control/qdb_query_server.log") or die
  "Cannot open server log for writing in append_to_log ... " .
    "exiting";

  $right_now = localtime;
  print MY_LOG "$right_now\n";
  print MY_LOG "$message\n";
  close(MY_LOG);
  return 1;
}

# The following function re-indexes the qcodes, it's placed here to shorten
# up the main part of the program.  Since it was pulled from the main
# program, it uses several global variables, these will be explicitly
# named with the package name.

sub reindex_qcodes {

  my(@dir_listing);
  my($directory);
  my($this_directory);
  my($this_access);
  my($most_recent);
  my($work_value);
  my($work_list);

  # Yet one more important less is to be had here.  I had set $/
  # (globally) to $CRLF in order to get 'standard' internet lines
  # from the socket.  Unfortunately, this separator is not used in
  # files, which this routine uses.  So it will reset it for the
  # duration of the routine, and set it back to 'normal' before
  # it exits.
  my($oldsep) = $/;
  $/ = "\n";

  # Get directory listing
  opendir(DB_DIR, $main::db_path) or
  append_to_log("Unable to open database directory ... exiting")
    and die "dying without grace.";
  @dir_listing = readdir(DB_DIR);
  closedir(DB_DIR);

  # Add new directory entries to the hash
  foreach $directory (@dir_listing) {
  if (!exists($main::qdb_index_last_access{$directory}) &&
    $directory ne "." && $directory ne ".." &&
    $directory ne "control" ) {
    $main::qdb_index_last_access{$directory} = 1;
  }
  }

  # Now that we have the database directory, search the relevant files
  # we will need to be indexing to get a date and time to be used for
  # comparison - and determine if we actually need to update our last
  # index.

  while (($this_directory, $this_access) =
     each(%main::qdb_index_last_access)) {
  # Find the most recently (relevant) modified file in this directory
  # Note:  It is much faster to get the modification age by the -M
  # file test, so that is how it will be done (it used to be done
  # by getting a stat on the open file, but this is unnecessary)
  $most_recent = time -
    (-M "$main::db_path/$this_directory/Connectivity.raw") * 86400;
  # This (slightly strange) formula will give the modification time
  # in terms of the system clock, not how long ago it was.
  $work_value = time -
    (-M "$main::db_path/$this_directory/Qcodes") * 86400;
  if ( $work_value > $most_recent ) {
    $most_recent = $work_value;
  }
  $work_value = time -
    (-M "$main::db_path/$this_directory/Original_structure.raw")
    * 86400;
  if ( $work_value > $most_recent ) {
    $most_recent = $work_value;
  }

  # Now, $most_recent refers to the update time of the most
  # recently updated of the 3 files.  The next step is to compare
  # this value with the value in the hash, if it is greater, we
  # will need to re-index this value.
  if ($most_recent > $this_access) {
    # First, update the last access time
    $main::qdb_index_last_access{$this_directory} = $most_recent;

    # Delete any previous Qcode information from hash
    delete($main::qdb_index_qcodes{$this_directory});

    # Get the Qcode file from the disk, and re-index it
    open(TEMP, "<$main::db_path/$this_directory/Qcodes") or
    append_to_log("Warning, could not open $this_directory/".
            "Qcodes for reading, daemon terminating") and
            die "Could not open a critical file, check log for details";

    @work_list = <TEMP>;
    chomp(@work_list);
    close(TEMP);

    # And, reconstruct it
    # Split the individual items, add them to the hash
    foreach $line (@work_list) {
    push(@{$main::qdb_index_qcodes{$this_directory}},
       [ split(/ /, $line) ]);
    }
    # This qcode has been reindexed.
  }
  }

  # While the following is commented out, it is still useful as a guide to
  # getting stuff back out of the qdb_index_qcodes hash

  # my($key, $value, $work_value2);
  # print "Qcode index times:\n";
  # while(($key, $value) = each(%main::qdb_index_qcodes) ) {
  #   print "$key:\n";
  #   foreach $work_value (@{$value}) {
  #     foreach $work_value2 (@{$work_value}) {
  #       print "$work_value2 ";
  #     }
  #     print "\n";
  #   }
  #   print "\n";
  # }

  # And, reset the input record separator.
  $/ = $oldsep;

}

# The following function outputs bond matches in the format:
# <directory_with_match> <atom1> <atom2> <deviance_of_1st_atom>,
#  <deviance_of_2nd_atom><tab>.
#
# It leaves the task of finishing the transmission (adding the $CRLF)
# to the main part of the program.  The list of items it takes has
# members formatted as follows:
# <directory> <atom1> <atom2>

sub output_bond_matches {
  my(@list_of_matches) = @_;
  foreach (@list_of_matches) {

  my ($directory, $atom1, $atom2) = @{$_};

  print $main::client "$directory ";

  if ($main::debug) {
    print "Sending to client:x$directory ";
  }

  # Within this loop, the reference to the qcode
  # list for atom 1 is given as:
  # $main::qdb_index_qcodes{@{$_}[0]}[@{$_}[1]]
  # And the reference to the second qcode is:
  # $main::qdb_index_qcodes{@{$_}[0]}[@{$_}[2]]
  # Finally, the reference to the queried qcode is:
  # \@main::queried_atom1_qcode and
  # \@main::queried_atom2_qcode
  # Ok, it seems that trying to actually write the
  # references in their native format looks pretty
  # much hopelessly obfuscated.  This small block
  # will set us up to call the get_qcode_deviance()
  # function.  Aside from that, we need to know
  # which qcode matched which atom (I mentioned
  # doing bond matches was a bit ... busy, didn't
  # I?

  # Note that depending on how the qcode matches, we may have to
  # send the atoms back in a different order than they're stored
  # ... we defer actually sending the atom numbers until we actually
  # have the proper order.

  my(@refs_to_pass) = ();

  push(@refs_to_pass, $main::qdb_index_qcodes
     {
      @{$_}[0]}[@{$_}[1]]);
  push(@refs_to_pass, $main::qdb_index_qcodes
     {
      @{$_}[0]}[@{$_}[2]]);
  push(@refs_to_pass, \@main::queried_atom1_qcode);
  push(@refs_to_pass, \@main::queried_atom2_qcode);

  # The following if statement is a long way
  # to say:  "Is the match we found between
  # 0-2 and 1-3, or 0-3 and 1-2?".  The numbers
  # refer to the references we record in the
  # series of push statements above.
  if (sqrt(CFUNCS::get_qcode_deviance($refs_to_pass[0],
                  $refs_to_pass[2]) *
       CFUNCS::get_qcode_deviance($refs_to_pass[1],
                  $refs_to_pass[3])) >
    sqrt(CFUNCS::get_qcode_deviance($refs_to_pass[1],
                  $refs_to_pass[2]) *
       CFUNCS::get_qcode_deviance($refs_to_pass[0],
                  $refs_to_pass[3]) ) ) {

    if ($main::debug) {
    print "$atom1 $atom2 ";
    print CFUNCS::get_qcode_deviance($refs_to_pass[0],
                   $refs_to_pass[2]);
    print " ";
    print CFUNCS::get_qcode_deviance($refs_to_pass[1],
                   $refs_to_pass[3]);
    print "x\n";
    print "Sending matches straight\n";
    }

    print $main::client "$atom1 $atom2 ";
    print $main::client
    CFUNCS::get_qcode_deviance($refs_to_pass[0],
                 $refs_to_pass[2]);
    print $main::client " ";
    print $main::client
    CFUNCS::get_qcode_deviance($refs_to_pass[1],
                 $refs_to_pass[3]);
  } else {

    if ($main::debug) {
    print "$atom2 $atom1 ";
    print CFUNCS::get_qcode_deviance($refs_to_pass[0],
                   $refs_to_pass[3]);
    print " ";
```

```
    print CFUNCS::get_qcode_deviance($refs_to_pass[1],
                   $refs_to_pass[2]);
    print "x\n";
    print "Sending matches reversed\n";
    }

    print $main::client "$atom2 $atom1 ";
    print $main::client
    CFUNCS::get_qcode_deviance($refs_to_pass[0],
               $refs_to_pass[3]);
    print $main::client " ";
    print $main::client
    CFUNCS::get_qcode_deviance($refs_to_pass[1],
                $refs_to_pass[2]);


  }
  if ( $_ != $list_of_matches[$#list_of_matches]) {
    print $main::client "\t";
  }
  }
}
```

qdb_local_submit.pl

```
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This is a 'daemon' that controls the submission of all new files.  It
# will (of course) rely on the .qdb_checkrc file for configurations.
# It will alternately read from the $db_path/control/que file (with
# appropriate responsibility towards file locking) check for resources,
# and submit jobs.  Also, it will check to see when jobs are done, and
# when they are, it will send a SIGALRM to the process that requested
# the job.  When it is running well and stable, it will simply run
# constantly on the intended machine, perhaps being set to restart from
# the crontab file.

# Since this was a pretty early program, I will _not_ be using taint
# checking and extensive security.  The worse case scenario concering
# this is that a malicious user could put a lot of jobs in the file,
# and have the computers working (up to their limit) until the problem
# is discovered.  This is more or less unavoidable, or perhaps I just
# haven't pondered different designs enough.  Regardless, we'll not worry
# about securing this daemon until there's more time.

# Another note:  After cleaning this up after adding the use strict
# pragma, I've come to the realization that it's quite messy.  Sorry about
# that, but once again, as long as it's working for our purposes, we'll
# leave it as is.

# There is a _major_ limitation with this program's implementation.
# It will happily submit jobs until there's no more resources, so if
# a program submits jobs repeatedly to the que, it will simply result
# in that job being run over and over again.  It's the responsibility
# of the program that writes to the que to only make one request for
# a given job.  This should definately be fixed in a later release.  After
# troubleshooting with some (of my own) programs, I've decided to
# do a bit of a kludge with the current setup.  If you call write list
# on an already locked file, it will happily replace the file with the
# contents of the list, and the filename should remain locked, though
# I'm not certain about this.  The program will be rewritten to maintain
# it's locks throughout.  Also, it will regularely check the jobnames
# in the que agains the job names in submitted jobs, and remove from the
# que any jobs that are mentioned in the submitted jobs.

BEGIN {
  # Since our own modules aren't properly installed, add to the INC
  # list at compile time
  push(@INC, "../perl_modules");
}

# Included libraries
require('../general/rc_file_handling.pl');
require('../perl_modules/local_functions.pl');
use NETFLOCK qw(:basic);

# pragmas
use strict;

# Function prototypes
sub write_list($@);

# This program will need handle signals.  The signals we will handle
# and their appropriate actions will be:
# SIGHUP    1    Ignored (Shell sends this signal when exiting)
# SIGALRM   14   Wake up (default handler)
# SIGTERM   15   Respawn ourself
# SIGSTOP   17   Ignore (Shell sends this signal when exiting)
# SIGQUIT   29   Exit gracefully

# The following subroutines are defined at the end of the program
use sigtrap 'handler', \&handle_hup, "HUP";
use sigtrap 'handler', \&handle_alrm, "ALRM";
use sigtrap 'handler', \&handle_term, "TERM";
use sigtrap 'handler', \&handle_stop, "STOP";
use sigtrap 'handler', \&handle_quit, "QUIT";

# Process the .qdb_checkrc file
open("RCFILE", '<.qdb_checkrc') or die
  "Unable to open .qdb_checkrc ... exiting\n";

my($db_path) = read_scalar("RCFILE", "db_path");
defined($db_path) or die
  "Unable to find db_path in .qdb_checkrc file ... exiting\n";

my(@hosts) = read_list("RCFILE", "available_hosts");
scalar(@hosts) or die
  "Unable to find hosts in .qdb_checkrc file ... exiting\n";

my($host_connect_method) = read_scalar("RCFILE", "host_connect_method");
defined($host_connect_method) or die
  "Unable to find host_connect_method in .qdb_checkrc file ... exiting\n";

my($local_ab_initio_program) = read_scalar("RCFILE",
"local_ab_initio_program");
defined($local_ab_initio_program) or die
  "Unable to find local_ab_initio_program in .qdb_checkrc file " .
  "... exiting\n";

$main::full_command_name = read_scalar("RCFILE", "submission_daemon_path");
defined($host_connect_method) or die
  "Unable to find host_connect_method in .qdb_checkrc file ... exiting\n";
my(@work_list) = split("\/", $0);
my($work_string) = pop @work_list;
$main::full_command_name .= "/$work_string";

# And load the appropriate ab_initio program specific function names
require("../perl_modules/${local_ab_initio_program}_functions.pl");

close(RCFILE);

# The que is in place, we need only create the working loop.  The basic
# algorithm is:
# 1)  Check $db_path/control/submitted_jobs, which will have the format:
#     <requesting_user>,<parent_host>,<parent_pid>,<job_host>,<job_cmd_line>
#     Note:  job_cmd_line is a bit deceptive, it's only the name of
#            the job that was run, not the whole command line.
# 2)  For each job, verify that it's running.  If it is not, it may be
#     finished.  Verify it is finished by looking at the resulting .log
#     file.  (Via a callout)  If it it not finished, place it at the
#     _beginning_ of the que file.  If it is finished go to step 3),
#     otherwise, move on to step 4).
# 3)  If the job is finished, remove that job from the que. Write the file
#    "$db_path/control/message.<parent_pid>" and wake the parent.
# 4)  Check the local system for computing resources.  If there are no
#     resources, sleep for one minute.  If there are, verify that the
#     job to be submitted has not previously finished (if it has, discard
#     it).  If it's not, submit the next job in the que, move it to
#     submitted_jobs, and sleep for one minute.  If there are no jobs to
#     run, sleep for another minute, and repeat the cycle.

# Note, I've implemented my own module for locking over an NFS
# network, since the Fcntl module does not do this.
# use Fcntl ':flock'; # import LOCK_* constants

# Open (and initialize) our errlog file.
open (ERR_LOG, ">$db_path/control/qdb_local_submit_err.log") or die
  "Cannot open error log for writing in qdb_local_submit.pl ... " .
  "exiting";
close(ERR_LOG);

my_warn("qdb_local_submit.pl staring up ...");
my_warn("My pid is $$\n");

# Create submitted jobs if it doesn't exist
unless ( -e "$db_path/control/submitted_jobs" ) {
  open(TMP_FILE, ">$db_path/control/submitted_jobs") or die
  "Unable to create submitted_jobs file in qdb_local_submit.pl" .
    "... exiting";
  close(TMP_FILE);
}

# Create an empty que if it doesn't exist
unless ( -e "$db_path/control/que" ) {
  open(TMP_FILE, ">$db_path/control/que") or die
  "Unable to create que file in qdb_local_submit.pl" .
    "... exiting";
  close(TMP_FILE);
}

# Get the current que.  There should _never_ be duplicate jobs submitted
# to the que, as doing so would result in not knowing who to wake when
# the job is done.  This is (was) a major limitation.  We will refuse
# to submit jobs that have the same input name as anything in the
# submitted jobs list.
my(@que_list) = ();
nflock("$db_path/control/que");
open (QUE, "<$db_path/control/que") or
  my_warn("Could not open the que for reading in intialization.  If " .
      "it doesn't exist because there are no requested jobs, this " .
      "will be ok.  Otherwise, it may be fatal.");
@que_list = <QUE>;
chomp(@que_list);
# Note that we do _not_ lock the que, unless we're using it.
# When it needs updating, it will be reopened as a write file and written.

nflock("$db_path/control/submitted_jobs");
open (SBMT, "<$db_path/control/submitted_jobs") or die "Could not open " .
  "the submitted jobs file for reading ... exiting\n";
```

```
my(@submitted_jobs) = <SBMT>;
close(SBMT);
chomp(@submitted_jobs);
# We _do_ leave the submitted jobs file locked, since this is the only
# process that should be changing it.
nfunlock("$db_path/control/submitted_jobs");

my($i);
my(%work_hash) = ();

# Check the que for jobs that already exist in the submitted jobs.
foreach ( @submitted_jobs ) {
  my($jobname);
  (undef, undef, undef, undef, $jobname) =
  split(",", $_);
  $work_hash{$jobname} = 1;
}
for ( $i = 0; $i <= $#que_list; $i++ ) {
  my($jobname);
  (undef, undef, undef, $jobname) = split(",", $que_list[$i]);
  if (exists($work_hash{$jobname})) {
  splice(@que_list, $i, 1);
  $i--;
  my_warn("Job $jobname found on the que, but was already " .
      "submitted.  It has been deleted (Note 1)");
  }
}

# Note that all file writes must be 'atomic'.  i.e., we write the target
# file to a temp file, add a link to that file with link(), then
# unlink() the temp file.  This guarantees that there will never
# be a partial, or mangled file.  The function write_list() accomplishes
# this (while attempting to preserve the lock on the original file).

# Before the submission begins, we need to do some cleanup in case we are
# recovering from a crash or restart.  The first order of business is to
# look for all of the jobs in the que and submiited jobs, and check on
# each of the machines for those processes being active.  If a job is found
# active, it's removed from the list.  After all of the machines have been
# checked, the files are written back.

# First, see if any of the sumitted jobs are dead.  If they are, shift
# them onto the @que_list.  Then, go on and check the que.

my($this_user, $parent_host, $parent_pid, $job_host, $job_cmd_line);
for ($i = 0; $i <= $#submitted_jobs; $i++) {
  ($this_user,$parent_host, $parent_pid, $job_host, $job_cmd_line) =
  split(",", $submitted_jobs[$i]);
  my($is_job_dead) = 1;
  for (@hosts) {
  my($this_host) = $_;
  if ( is_running($this_host, $job_cmd_line) ) {
    $is_job_dead = 0;
    last;
  }
  }
  if ($is_job_dead) {
  my_warn("Dead job $job_cmd_line found in submitted_jobs, " .
      "putting it at the beginning of the que");
  # Put it on the que (if it's not already there)
  splice(@submitted_jobs, $i, 1, () );
  unless (grep (/$job_cmd_line/, @que_list) ) {
    unshift(@que_list, "$this_user,$parent_host," .
        "$parent_pid,$job_cmd_line");
    unless ($i == 0) {
    $i--;
    }
  }
  }
}

# And, write the files back
write_list("$db_path/control/que", @que_list);
write_list("$db_path/control/submitted_jobs", @submitted_jobs);


# Now check the que list
my($this_pid, $this_job);
for ($i = 0; $i <= $#que_list; $i++) {
  ($this_user,$work_string,$this_pid,$this_job) =
  split(",", $que_list[$i]);
  my_warn("In initializtion, checking for presense of job $this_job" .
      "(" . ($i + 1) . "/" . ($#que_list + 1) . ")\n");
  for (@hosts) {
  my($this_host) = $_;
  if ( is_running($this_host, $this_job) ) {
    my_warn("Found the job $this_job already running on host " .
        "$this_host, placing on the submitted jobs list");
    splice(@que_list, $i, 1, () );
    push(@submitted_jobs, "$this_user,$this_host,$this_pid," .
       "$this_host,$this_job");
    unless ($i == 0) {
    $i--;
    }
  }
  }
}

# And, write the que back (in case it was changed)
write_list("$db_path/control/que", @que_list);
write_list("$db_path/control/submitted_jobs", @submitted_jobs);

# Start the forever loop.  This program is never intended to die.  Note
# that the variable $job_cmd_line is somewhat of a misnomer, as it's
# really just the name of the input file, and the command line is
# built from it.
$main::run_forever = 1;
while ($main::run_forever) {

  my_warn("Beginning main loop");

  # Open and lock the QUE while we're processing
  nflock("$db_path/control/que");
  open (QUE, "<$db_path/control/que") or
```

```
my_warn("Could not open the que for reading after sleep.  If it " .
    "doesn't exist because there are no requested jobs, this " .
    "will be ok.  Otherwise, it may be fatal.");
@que_list = <QUE>;
chomp(@que_list);

%work_hash = ();

# Check the que for jobs that already exist in the submitted jobs.
foreach ( @submitted_jobs ) {
my($jobname);
(undef, undef, undef, undef, $jobname) =
  split(",", $_);
$work_hash{$jobname} = 1;
}
for ( $i = 0; $i <= $#que_list; $i++ ) {
my($jobname);
(undef, undef, undef, $jobname) = split(",", $que_list[$i]);
if (exists($work_hash{$jobname})) {
  splice(@que_list, $i, 1);
  $i--;
  my_warn("Job $jobname found on the que, but was already " .
      "submitted.  It has been deleted (Note 2)");
}
}

# Loop through the jobs and see if they're done
for ($i = 0; $i <= $#submitted_jobs; $i++) {
($this_user,$parent_host, $parent_pid, $job_host, $job_cmd_line) =
  split(",", $submitted_jobs[$i]);

unless ( is_running($job_host, $job_cmd_line) ) {
  # Now, check to see if the job is done:  If it is, inform
  # the parent, and move on, if it's not, place it at the
  # beginning of the que
  if (is_finished($job_cmd_line) ) {
  # We need to use splice to remove this element, and
  # then decrement $i.
  splice(@submitted_jobs, $i, 1, () );
  write_list("$db_path/control/submitted_jobs",
         @submitted_jobs);

  # Write the message file
  nflock("$db_path/control/message.$parent_pid");
  open(TMP_FILE, ">>$db_path/control/message.$parent_pid")
    or my_warn ("Cannot open message file for writing, " .
        "this is likely to be fatal\n");
  print TMP_FILE "$job_cmd_line\n";
  close (TMP_FILE);
  nfunlock("$db_path/control/message.$parent_pid");

  # And inform the parent
  system("ssh -n $parent_host kill -s SIGALRM $parent_pid");

  $i--;
  } else {

  push(@que_list, "$this_user,$parent_host," .
      "$parent_pid,$job_cmd_line");
  my_warn("Job $job_cmd_line was found in the submitted " .
      "jobs list in main loop, but was dead.  It has " .
      "been returned to the end of the que.  It was  " .
      "suposed to be running on $job_host");
  # Make sure we didn't enter a duplicate job:
  @work_list = @que_list;
  my($j);
  for ($j = 1; $j <= $#work_list; $j++) {
    if ($work_list[0] eq $work_list[$j]) {
    my_warn("Job just returned to the que was " .
        "a duplicate!  Removing it from the " .
        "que");
    pop(@work_list);
    last;
    }
  }
  @que_list = @work_list;

  # Now write the file back into que
  write_list("$db_path/control/que", @que_list);

  # And finally, remove it from submitted_jobs and update
  splice(@submitted_jobs, $i, 1, () );
  write_list("$db_path/control/submitted_jobs",
         @submitted_jobs);

  $i--;
  }
}
}

# Now, check the system for resources, if there are any, go on with
# the submission.  Note that we need to check this with the
# next line of the que file, since we don't know what user is
# requesting the resources.
if ($#que_list >= 0) {     # If the list isn't empty
my($this_host);
my($least_loaded);
($this_user,$this_host,$this_pid,$this_job) =
  split(",",$que_list[0]);

$least_loaded = check_system_resources($this_user, @hosts);

if ($least_loaded ne "") {
  # In any case, shift the first value off the que (it has
  # already been recorded) and update the que file
  shift(@que_list);
  write_list("$db_path/control/que", @que_list);

  # Verify that the job has not been previously finished, if
  # it has, remove it from the que, if not, submit the job.
  unless ( is_finished($this_job) ) {

  # Submit it!
  push(@submitted_jobs, "$this_user,$this_host,$this_pid," .
```

```perl
      "$least_loaded,$this_job");
    write_list("$db_path/control/submitted_jobs",
          @submitted_jobs);

    my_warn("Submitting $this_job to $least_loaded for $this_user");
    submit_job("$this_user", "$least_loaded", "g98 $this_job");
    } else {
    # Inform the parent, and write the message file.
    nflock("$db_path/control/message.$this_pid");
    open(TMP_FILE, ">>$db_path/control/message.$this_pid")
      or my_warn ("Cannot open message file for writing, " .
          "this is likely to be fatal\n");
    print TMP_FILE "$this_job\n";
    close (TMP_FILE);
    nfunlock("$db_path/control/message.$this_pid");

    # And inform the parent
    system("ssh -n $this_host kill -s SIGALRM $this_pid");

    }
  }
  }

  # And unlock the que while we rest.  It may be that as soon as we
  # unlink the original que in a call to write_list(), we lose our
  # previous lock.  I don't know if this is true or not.  If someone
  # ever works this out, we may need to make write list simply work
  # with open and locked filenames.  I have been experiencing some
  # jobs that I believe were submitted, but in fact did not actually make
  # it to being calculated.  It may be because of this problem.

  open(QUE, "<$db_path/control/que");
  close(QUE);
  nfunlock("$db_path/control/que");

  sleep (180);

}

# If we ever get to this exit, we must have gotten a SIGQUIT (the previous
# while loop is not designed to exit
my_warn("Control reached end of program, indicating a normal exit.  " .
    "(At the mercy of a SIGQUIT\n");
nfunlock("$db_path/control/que");
nfunlock("$db_path/control/submitted_jobs");

exit 0;

########################################################################
#       End of program                                                 #
########################################################################

# Functions

sub my_warn {
  my($message) = join("",@_);
  # Open our errlog file.
  open (ERR_LOG, ">>$db_path/control/qdb_local_submit_err.log") or die
  "Cannot open error log for writing in my_warn ... " .
    "exiting";

  my($right_now);
  $right_now = localtime;
  print ERR_LOG "$right_now\n";
  print ERR_LOG "$message\n";
  close(ERR_LOG);
  return;
}

# The following routine simply does an atomic write of the list
# provided with the second (and subsequent) arguments to the file
# given by the first argument.  The function was originally designed
# to write the list into a temp file, and then link the new file, and
# unlink the temp file.  Unfortunately, this does not work in an NFS
# environment.  Neither does the flock() set of functions.  I've
# written my own module to handle this.  This function will obtain a
# lock on the given file, and will _not_ release it, as that
# responsibilty will lie with the calling environment.  This will keep
# the file locked for as long as this program has control.

sub write_list ($@) {

  my($filename) = shift;
  unless (defined($filename)) {
  my_warn("No filename provided to function write_list, this is " .
      "likely fatal");
  }
  my(@list_to_write) = @_;

  nflock($filename);

  open(TMP, ">$filename") or my_warn("Unable to open $filename for " .
                 "writing in write_list().  This will " .
                 "most likely be fatal");

  print TMP join("\n", @list_to_write);
  if ($#list_to_write >= 0) {
  print TMP "\n";
  }

  return;
}

sub handle_hup {
  # This handler does exactly nothing, exept reports the receipt of the
  # signal to the error_log file
  my_warn("Signal HUP received, ignoring");
  return;
}

sub handle_alrm {
  # This handler also does nothing, but records the wake up call,
  # and continues on it's merry way.  If the que is not empty, that
  # portion will change to a shorter sleep interval until there are
  # no resources left.
  my_warn("Signal ALRM received, waking up");
  return;
}

sub handle_term {
  # This handler simply respawns the daemon
  my_warn("Signal TERM received, respawning daemon");
  exec($main::full_command_name);
}

sub handle_stop {
  # This routine also ingores the signal, but appends it to the error_log
  my_warn("Signal STOP received, continuing");
  return;
}

sub handle_quit {
  # This is the "correct" way to terminate the daemon.  It will change
  # the forever loop variable (note that it's a global variable) to
  # 0, and the next pass through the loop will be it's last.
  my_warn("Signal QUIT received, marking variable to exit normally");
  $main::run_forever = 0;

  # Wake us in case we're in the sleep cycle
  system("kill -14 $$");
  return;
}
```

qdb_input_server.pl

```perl
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This program was originally copied from the final version of
# qdb_calculate.pl, as it's designed to take the place of that program
# (qdb_calculate has been moved to the graveyard, and is no longer in
# service).  Its responsibilities and design, however, have been largely
# expanded and specified, so it should be both more flexible, and
# powerful.  Here in the intro, I try to lay out all characteristics,
# responsibilities, and structure requirements for the program.


# Characteristics:

# - This program is a daemon, and is meant to run constantly.  Also
# there should only be one copy of it running for any given
# filesystem.

# - Upon startup, and continuously while it's running, it will keep
# all of the information it needs to 'recover' in a recovery directory

# - The daemon will use signals and files for interprocess
# communication, since it has no demand for fast 2-way communication

# - The (originally proposed) directory structure for the daemon are:

# $db_path/control/qis
# $db_path/control/qis/recover
# $db_path/control/qis/input
# $db_path/control/qis/output
# $db_path/control/qis/in_progress
# $db_path/control/qis/tmp
# $db_path/control/qis/junkyard

# - The daemon also (upon startup) registers it's pid and host in
# qdb_input_server.pid and qdb_input_server.host respectively

# - The program will be reasonably secure, and since it has no sockets
# to the world, the only damage a potentially malicious party could do
# is limitied to what someone with shell access can do.  This means
# that until the program has had a serious security audit, it should
# not be called as a result of any external call (i.e., from a web
# page, or whatnot).  Note that this doesn't mean I'll be ignoring
# security.  Once the program is started, it will set it's own home
# directory to $db_path, (mainly) limiting it's domain of damage to
# the database.  It does, however, make callouts (via ssh in the
# caller's environment) to other systems that can do ab initio
# calculations; these calls are subject to moderate scrutiny in the
# original development, and need a full audit before the program is
# allowed to link to the outside world.

# - A design goal (that I've largely been ignoring with my perl
# programs until now) is to keep as much of the 'crunching' out of the
# main package as possible.  This will add to the readability of main.

# - All user functions must be prototyped
```

```
# Responsibilities:

#  - The program is responsible for processing all of qdb_check's
# output.

# - The program is responsible for formatting and submitting all
# initial calculations on new fragments from qdb_check's output.  As
# such, it also needs to run
# ff/qdb/qdb_maintenance_utilities/utilities/determine_frozen_bonds
# for each new database directory.  Finally, it needs to notice when
# these calculations are done, and (atomically) enter the new
# directory.

# - The program is responsible for starting the torsion calculation
# daemon for each requested torsion, noting the finish times for them,
# and entering the results in the appropriate directory.

# - The program is responsible for starting the torsion_ft_fit
# program, with appropriate arguments.  This is no longer true!  The
# torsion scanning daemon will also try to get the best fit, and enter
# it into the database fragments proper torsion subdirectory.

# - Finally, after all of these requests are filled, it starts the
# final force field collator

#  Structure:

# - Initialization

#  - reads .qdb_checkrc variables - register it's own host and pid,
#  fail to startup if there's already one running
#  - builds $db_path/control/qis directory structure if needed

#  - start main loop
#   - Main loop
#    - sleep
#    - when awakened:
#    - re-initialize all relevant areas
#    - read contents of input directory
#    - process request
#    - when all requests are finished, go back to sleep

# Important Files:

# - Some files will be static, and they are used as both checkpoint
# information, and can be referred to from anywhere to learn about
# certain states.  They are as follows:

#    $db_path/control/qis
#    $db_path/control/qis/recover
#    $db_path/control/qis/input
#    $db_path/control/qis/output
#    $db_path/control/qis/in_progress
#    qdb_entries_in_progress <-- Contains the names of qdb_entries
#                                we-recurrently working on, and their
#                                full path prefix (separated by a space)
#    $db_path/control/qis/tmp
#    $db_path/control/qis/junkyard


# Other notes:
# The program will not use file locking, since it's the only program
# 'allowed' to make changes to the database.  The exception to this
# rule is that the que will need to be accessed with 'proper' file
# locking semantics.  It is also responsible for following file
# locking semantics for items in the input directory.
# I come to a bit of a quandry in development.  I don't see any simple
# way to atomically read and write directories, so I'll go for the next
# best thing.  Each database directory will have a 0 length file named
# is_hosed place in it as soon as the directory is created.  The database
# verification utility will check for this file, and note it in it's
# report.  The file will be removed after the directory is fully
# entered.

# Given the extent of list and file manipulation this program does, it
# may be prudent at some point in the future to surround sensitive
# sections of code with function that indicate (on disk) that the program
# crashed.  This could be as simple as creating a file, and deleting
# it when we're finished.  The daemon would then, upon startup, check to
# see if they do exist, and do a cleanup before actually restarting.
# I don't anticipate a lot of problems with crashing, but it might be
# a nice feature to implement in the future.

# Note that since this program uses 'standard' flock file locking, it
# is vulnerable to being 'fooled' when it's run on a machine different
# than qdb_local_submit.pl.  For this reason, it should be run on the
# same machine.

package main;

use strict;
use sigtrap 'handler', \&wake_up, "ALRM";
use sigtrap 'handler', \&handle_hup, "HUP";
use sigtrap 'handler', \&quit_now, "QUIT";
use Cwd;
use Fcntl ':flock';        # import LOCK_* Constants

# use File::Path;    <-- rmtree is broken with respect to taintedness.
#                        When it's fixed, or when someone wants to
#                        write their own solution, use that ... until
#                        then, simply use the shell.

eval { require 5.6.1 }
  or die <<MESSAGE;
###################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
###################################################################
MESSAGE

# Before proceeding, clean up our environment so we can run external
# programs
require '../general/clean_environment.pl';
full_env_clean();

# Declare global variables.  Note that these will be global to the entire
# file
my($db_path);          # The functions need to know where it is
my($loop_forever) = 1;      # Signal handlers manipulate this
my($got_alarm) = 1;       # Signal handlers manipulate this
my($debug) = 0;    # Set this to non-zero to see extra debugging info
my($ab_initio_program);
my($ab_initio_suffix);
my($host_connect_method);
my($this_user);
my(%qdb_entries);
my($starting_directory) = getcwd;
my($this_host) = $ENV{"HOSTNAME"};
my(%completed_requests) = ();

# Launder starting directory.  It _must_ be the directory we started in,
# otherwise the program will die when it tries to read .qdb_checkrc.
($starting_directory) =
  $starting_directory =~ m%([\w/]+)%; # %'s used since / is in the pattern.

# Launder $this_host ... if someone messes with HOSTNAME in the environment,
# the only ill effect is that the proper host won't get a SIGALRM when the
# calculation is finished ... that's pretty innocuous, huh?
($this_host) = $this_host =~ m%([\w.]+)%;

# Function prototypes
sub get_CHNO_formula (\@);
sub append_to_log(@);       # Gobble all arguments for log addition
sub wake_up;
sub handle_hup;
sub quit_now;
sub get_database_entries;
sub read_molecule(\@);
sub print_molecule(@);
sub read_qdb_input_list(\@);
sub is_molecule_unique($@);
sub build_submit_scaffold_dir($\@);
sub get_unique_name_in_dir ($$;$@);
sub move_flat_directory ($$);
sub remove_message ($);
sub change_frag_to_qdb_in_list($\@\@);
sub is_CHNO_name ($);
sub update_job($\@\@\@);
sub print_map_list (@);
sub start_needed_torsions(\@\@);

# Begin processing .qdb_checkrc file
require "../general/rc_file_handling.pl";

open("RCFILE", '<.qdb_checkrc') or die
  "Unable to open .qdb_checkrc ... exiting\n";

$db_path = read_scalar("RCFILE", "db_path");
defined($db_path) or die
  "Unable to find db_path in .qdb_checkrc file ... exiting\n";

# Since we trust the contents of .qdb_checkrc, launder $db_path
($db_path) = $db_path =~ m%([\w/\.]+)%; # %'s used since / is in the pattern.

$ab_initio_program = read_scalar("RCFILE", "local_ab_initio_program");
defined($ab_initio_program) or die
  "Unable to find ab_initio_program in .qdb_checkrc file ... exiting\n";

# And, launder it
($ab_initio_program) = $ab_initio_program =~ m/([\w]+)/;

$ab_initio_suffix = read_scalar("RCFILE", "ab_initio_suffix");
defined($ab_initio_suffix) or die
  "Unable to find ab_inition_suffix in .qdb_checkrc file ... exiting\n";

$this_user = read_scalar("RCFILE", "user_name");
defined($this_user) or die
  "Unable to find user_name in .qdb_checkrc file ... exiting\n";

close("RCFILE");

# Register our pid and host
{
  # Do it in its own block since we can discard the variables when we're
  # done with them

  ( my($host), undef ) = split(/\./, $ENV{"HOSTNAME"}, 2);
  open(TMP, ">$db_path/control/qdb_input_server.host") or die
  "Unable to register my hostname, exiting\n";
  print TMP "$host\n";
  close(TMP);
  chmod(0664, "$db_path/control/qdb_input_server.host");

  open(TMP, ">$db_path/control/qdb_input_server.pid") or die
  "Unable to register my pid, exiting\n";
  print TMP "$$\n";
  close(TMP);
  chmod(0664, "$db_path/control/qdb_input_server.pid");
}

# Note our startup in the log
append_to_log("qdb_input_server starting up");

# Check our directory structure and create it if needed
{
  # Again, enclosing block because we can discard any variables after
  # usage
  my(@dir_list) = (
          "$db_path/control/qis",
          "$db_path/control/qis/recover",
          "$db_path/control/qis/input",
          "$db_path/control/qis/output",
          "$db_path/control/qis/in_progress",
```

```
          "$db_path/control/qis/tmp",
          "$db_path/control/qis/junkyard",
         );

  for (@dir_list) {
  if ( -e $_ ) {
    unless ( -r $_ and -w $_ and -d $_ and -x $_ ) {
    unlink($_) or die "Unable to remove $_, as it wasn't a " .
      "proper directory, exiting\n";
    }
  } else {
    mkdir($_) or die "Unable to create directory: $_, exiting\n";
    chmod(0775, $_) or die "Unable to change permissions on " .
    "newly created directory ... exiting\n";
  }
  }
}

########################################################################
#                           Begin program                              #
########################################################################

# Reset the alarm, and start it
my($alarm_timeout) = 600; # How often do we do general checking of things?
              # Note that we should never actually have to
             # check anything with this timer, but this is
              # here to handle the odd case the I missed
              # some logic in receiving signals from other
              # sources

# Change our directory to qis, and never look back
chdir("$db_path/control/qis") or die
  "Unable to change to input server's home directory " .
  "($db_path/control/qis)";

MAIN_LOOP: while ($loop_forever) {

  # Commonly reused loop variables
  my(@work_list);
  my(%processed_jobs);

  # And restart the alarm
  alarm($alarm_timeout);

  # Sleep until we're woken
  sleep unless $got_alarm;

  # From this point on, all exits should also be noted in the log

  # Before checking what kind of job we might have, we need to look for a
  # message file under our name, and deal with any messages we see there.
  # These message files are written by qdb_local_submit.pl, to whom we'll
  # also be submitting jobs to in a semi system independant fashion.

  my($is_file_open) = 0;

  open(TMP_FILE, "<$db_path/control/message.$$") and $is_file_open = 1;
  if ($is_file_open) {
  my($jobname);
  @work_list = ();
  flock(TMP_FILE, LOCK_EX);
  @work_list = <TMP_FILE>;
  flock(TMP_FILE, LOCK_UN);
  close (TMP_FILE);
  foreach (@work_list) {
    chomp;
    $completed_requests{$jobname} = 1;
  }
  # These messages will remain in the control directory until they
  # have been dealt with specifically.
  }

  # Grab the next job from the input directory.  Note that jobs will
  # be grabbed one at a time, and processed, until the directory is
  # empty.  At that point, set $got_alarm to 0 and go back to sleep
  opendir(TMP_DIR, getcwd . "/input") or
  append_to_log("Unable to get directory listing from " .
        "input, server closing down")
    and die "Unable to read in critical directory, see log for details";

  my($this_job_filename) = undef;
  while ( defined($this_job_filename = readdir(TMP_DIR)) and
  $this_job_filename =~ /[.]{1,2}/ and
  !exists($processed_jobs{$this_job_filename}) ) { }

  closedir(TMP_DIR);

  # If we didn't find anything, move on sleepily
  unless ( $this_job_filename ) {
  $got_alarm = 0;
  next MAIN_LOOP;
  %processed_jobs = ();
  }

  $processed_jobs{$this_job_filename} = 1;

  # And don't sleep till we run out of filenames
  $got_alarm = 1;

  # Launder $this_job_filename, since we need it for future work
  unless( $this_job_filename =~ m/([\w.]+)/ ) {
    append_to_log("Illegal filename $this_job_filename read from " .
        "input directory, skipping this query");
  next MAIN_LOOP;
  }
  $this_job_filename = $1;

  if ( $this_job_filename ) {

  # We have a job to do!  Do it.
  open(TMP, "<input/$this_job_filename" ) or
    append_to_log("Unable to open " .
        "$db_path/control/qis/input/$work_list[0] for reading " .
        "in main loop, this is a critical error ... exiting")
    and die "Critical file error, see log for details";

  # Lock it before reading from it
  flock(TMP, LOCK_EX);

  # Note:  The next line may be slightly dangerous.  If a malicious
  # user puts a large file in the input directory, it will consume
  # a large portion of RAM, and may even put the system on a swap
  # fest until the systems kills this process.  This danger is mitigated
  # by system set ulimit's.  If someone cares to make it a bit more
  # secure, feel free to add a file size test before actually opening
  # the file.
  my(@current_job) = <TMP>;
  foreach (@current_job) {
    chomp;
  }

  flock(TMP, LOCK_UN);
  close(TMP);

  # The types of jobs the server can do are enumerated here:
  # $current_job[0] =~ m/^Begin parent molecule:&/
  #  This is output from qdb_check, and requires the following:
  #    - Format and submit all fragments for calculation.
  #    - Enter fragments that have been finished into the
  #      database, and change the job description to indicate
  #      the fragment is in the qdb now.
  #    - Submit requests for all torsions.  Finished torsions
  #      should be marked in the job description file (maybe by
  #      appending a + or something?)
  #    - Check the %completed_requests hash for instructions
  #      relevant to us
  #    - When all calculations are finished, send an e-mail to the
  #      user, indicating the job is finished, and ready to be
  #      collated
  # $current_job[0] =~ m/^Torsion scan complete&/
  #  This is output from the torsion scanning daemon, and requires the
  #  following
  #    - Note that the second line of the file should have the
  #      names of the two atoms for which the torsion was done
  #    - Append to message.pid file, and restart the main loop cold
  #         (with $got_alarm = 1).  This is necessary, since the
  #      job that cares about it is one of the outputs from qdb_check.
  #      Additionally, only we know our PID.


  # See what kind of job we got.
  if ($current_job[0] =~ m/^Begin parent molecule:$/) {
    # We're processing output from qdb_check

    # Note:  The following function munches @current_job up to
    # the end of the next molecule section, this is why we saved
    # the copy in the last step.  The other functions we call
    # have similar behavior
    my(@parent_molecule) = read_molecule(@current_job);
    my(@atom_map_list) = read_qdb_input_list(@current_job);
    my(@bond_map_list) = read_qdb_input_list(@current_job);

    # Now ... get all of the rest of the fragments (if there are
    # any to be had
    my(@fragment_list) = ();
    my(@tmp_mol);

    while ( @tmp_mol = read_molecule(@current_job), $tmp_mol[0]) {
    # Wow, so many important lessons.  We can't simply push
    # \@tmp_mol onto the list, because then the list just
    # references the variable (*doh*!)
    push(@fragment_list, [ @tmp_mol ]);
    }

    # Ok, we're fully initialized, now we need to do several tasks.
    # First, we build a 'pseudo qdb database entry' for each of the
    # new fragments.  We can then add our request to the que.
    # (We'll be taking advantage of what qdb_local_submit.pl already
    # does on our own system).
    # Once we know that, we can happily do all of the file moving
    # and such.

    # Before we build any scaffold directories, we need to check on
    # the fragments.  This section does it's best to prevent
    # duplicate entries in the database.  We'll check in three places
    # to see if the fragment we're working on is a duplicate of
    # any fragments there.  Note that the criteria for duplicate is
    # not any complicated bond matching, but much more simple.  If
    # the molecules have the same molecular formulae, and identical
    # qcodes (in any order), they are considered duplicates.  As I
    # comment in the function that checks them, I don't know if this
    # is rigorously true, but I'll assume it is until further
    # notice.
    #
    # What do we do if we find duplicates in various places?
    # $db_path           <- Duplicate fragment submitted, move the
    #                       request to the junkyard (this shouldn't
    #                       happen if qdb_check made the request)
    # $this_job_filename <- This is perfectly ok, it means we're
    #                       recovering from a crash ... don't
    #                       rebuild or re-submit the query, just
    #                       ignore it.  It's possible that if the
    #                       whole _system_ crashed, it wouldn't
    #                       be in the que, in which case the process
    #                       should be restarted from the beginning
    # <another directory <- This would be bizzare indeed, but quite
    #  in in_progress>      possible if the user was submitting
    #                       multiple similar jobs.  If this happens,
    #                       just move the job to the junkyard, and
    #                       request the user to try the full query
    #                       later (when the first is finished)

    # Prepare the in_progress directory list

    @work_list = glob("$db_path/control/qis/in_progress/*");

    # Remove qdb_entries_in_progress and exteraneous entries
    # from this list;
```

```
  my(%work_hash) = ();

  foreach ( @work_list ) {
  $work_hash{$_} = 1;
  }

  delete( $work_hash{"qdb_entries_in_progress"} );
  delete( $work_hash{"."} );
  delete( $work_hash{".."} );

FRAGLOOP: foreach(@fragment_list) {
  my(@this_frag) = @{$_};
  foreach (keys(%work_hash)) {
    if ( $_ =~ /$this_job_filename$/ ) {
    unless ( is_molecule_unique ( "$_", @this_frag ) ) {
      # This is ok, we just don't do anything in this
      # case
      next FRAGLOOP;
    }
    } else {
    unless ( is_molecule_unique ( "$_", @this_frag ) ) {

      # This is the case that this fragment was found
      # in another in_progress directory, we need to
      # move it to the junkyard and note this in the
      # log

      my($oldname, $newname);
      $oldname = $this_job_filename;
      $newname = get_unique_name_in_dir
      ($oldname, "$db_path/control/qis/junkyard");

      link("input/$oldname", "junkyard/$newname") or
      append_to_log("Unable to create new link " .
              "for $oldname, this is a critical " .
              "error, exiting")
        and die "Critical file error, see log " .
              "for details";

      unlink("input/$oldname") or
      append_to_log("Unable to create unlink " .
              "$oldname, this is a critical error, " .
              "exiting") and
              die "Critical file error, see log for details";

      append_to_log("An identical fragment was found " .
            "in the in_progress directory.  In order to " .
            "prevent adding duplicate fragments to the " .
            "database, it has been moved to the junkyard " .
            "with filename $newname");

      # Finally, we need to remove any trace of this
      # job from qdb_entries_in_progress

      my($is_open) = 0;
      open(TMP, "<in_progress/qdb_entries_in_progress")
      and $is_open = 1;

      if ($is_open) {
      # It's open.
      @work_list = <TMP>;
      close(TMP);
      my(%tmp_hash);
      foreach (@work_list) {
        $tmp_hash{$_} = 1;
      }

      foreach ( @work_list ) {
        if ( ($_) =~ m|$oldname/$| ) {
        delete($tmp_hash{$_});
        }
      }
      @work_list = keys(%tmp_hash);

      # Write the file back out
      open(TMP,
         ">in_progress/qdb_entries_in_progress") or
           append_to_log("Unable to open input/" .
                "qdb_entries_in_progress for writing " .
                "in main().  This is a critical file " .
                "error, exiting") and
             die "Critical file error, see log for " .
               "details";
      print TMP join("\n", @work_list) . "\n";
      close(TMP);

      # And finally, we need to delete the entire
      # job tree.  The module function rmtree
      # would be fine as long as it untainted.
      system ("rm -rf " .
          "$db_path/control/qis/" .
          "in_progress/$this_job_filename");

      # And get out of this horridly deep logic
      next MAIN_LOOP;
      }
    }
    }
  }

  unless ( is_molecule_unique ( $db_path, @{$_} ) ) {
    # If it's in the database, qdb_check probably screwed up,
    # or there's a logical error in this program.  In either
    # case, we quit.  This is a pretty critical logical
    # error, and will result in the server dying without
    # any cleanup

    append_to_log("An identical fragment was found " .
          "in the quantum chemistry database.  This condition " .
          "indicates a logical error in either the submission " .
          "(someone submitted a duplicate fragment), or in " .
          "this program itself.  In either case, this is a " .
          "critical error, exiting");
    die "Critical logic error encountered in main, see " .
      "log for details";

  }
  build_submit_scaffold_dir($this_job_filename, @this_frag);
  }

  # Phew!  That last section was a handful!  The logic seems to
  # be working fine for now, so we can move on

  # The fragments have been submitted, and our next task is to
  # start torsion drivers for any torsions that we need.  Clearly,
  # we can only start this task for the items that are already
  # in the database, but we'll be moving elements from having
  # a fragment ID to having a qdb_id once their calculations are
  # finished.  Once again, if we need to recover, it's not a
  # problem, since the torsion driver will not re-calculate a
  # bond that's already been finished.

  @work_list =
  glob("$db_path/control/qis/in_progress/$this_job_filename/*");

  my(%tmp_hash);
  foreach (@work_list) {
  if ($_ !~ /[.]{1,2}/) {
    $tmp_hash{"${_}/Initial_optimization.com"} = 1;
  }
  }

  # Look for completed requests in that hash.
  foreach ( keys(%tmp_hash) ) {
  if (exists($completed_requests{$_})) {
    # This means the database has finished it's work, and we're
    # ready to move the directory in question to the database
    my($source_dir, $target_dir, $barename);
    my($message) = $_;

    @work_list = split("/", $message);
    pop(@work_list);
    $barename = $work_list[$#work_list];
    $source_dir = join("/", @work_list);
    $target_dir = "$db_path/$barename";

    move_flat_directory($source_dir, $target_dir);
    remove_message($message);

    # The (almost) last step is to change the id's in our
    # @atom_map_list and @bond_map_list 's to reflect the
    # change in location of the data
    change_frag_to_qdb_in_list($barename, @fragment_list,
              @atom_map_list);
    change_frag_to_qdb_in_list($barename, @fragment_list,
              @bond_map_list);

    # And finally, re-save our job, so we're prepared to
    # recover.
    update_job($this_job_filename, @parent_molecule,
        @atom_map_list, @bond_map_list, @fragment_list);
  }
  }

  # We have only a few more tasks to do before we are done with
  # handling this kind of file.

  # Start calculations on all torsions that are needed and not
  # started (as indicated by a - in the last place of the map list)
  # Finally, update the job.
  start_needed_torsions(@bond_map_list, @parent_molecule);

  update_job($this_job_filename, @parent_molecule,
      @atom_map_list, @bond_map_list, @fragment_list);

  # Check %completed_requests for messages from torsion_driver.pl,
  # which will look like:
  # $db_path/<CHNO_directory> <atom1> <atom2>.
  # If any requests are found, update the map
  # list to indicate that they are no longer in progress, but
  # finished (indicated by a + instead of a - )
  foreach ( keys(%completed_requests) ) {
  ######################################################
  #          BEGIN RE-DEVELOPMENT HERE                 #
  ######################################################
  }

  die "Development here can continue once a full torsion is available " .
    "in the database.  We'll simulate the message.$$ file (by " .
    "reading a message.123 file or something), and make sure we " .
    "can re-mark the end of the appropriate line to a +.  At that " .
    "point (once that's working) we can verify the cleanup steps, " .
    "and cut this daemon loose.  Note:  Line above should be lines.  " .
    "It's possible, and even likely, that there will be several " .
    "parent bonds that correspond to identical fragment bonds";

  print "Bond map list coming:\n";
  print join("\n", @bond_map_list) . "\n";

  # Finally, check our atom map list, and our bond map list.  If
  # all of the information we require is available, put the output
  # into the output directory (under the same job name, potentially
  # with something appended to it), remove $this_job_filename from
  # the input directory, e-mail the user, and
  # start the collator.


  # Start here;
  die "Almost finished!";

} elsif ($current_job[0] =~ m/whatever/ ) {

} else {

  my($oldname, $newname);
```

```
    $oldname = $this_job_filename;

    $newname = get_unique_name_in_dir($oldname,
                    "$db_path/control/qis/junkyard");

    link("input/$oldname", "junkyard/$newname") or
    append_to_log("Unable to create new link for $oldname, " .
            "this is a critical error, exiting")
      and die "Critical file error, see log for details";

    unlink("input/$oldname") or
    append_to_log("Unable to create unlink $oldname, " .
            "this is a critical error, exiting")
      and die "Critical file error, see log for details";

    append_to_log("Unrecognized job request in file $this_job_filename " .
          "moving it to the junkyard under name $newname");
  }
  }

  # Remove this line at end of development
  $loop_forever = 0;
}

print "Exiting after forever loop is finished\n";
exit;

$debug = 0;

exit;

########################################################################
#        End of program                                                #
########################################################################

# The following function scans through the bond_map_list provided, and
# submits any torsions that haven't either been submitted already, or
# exist only in frag_list
sub start_needed_torsions(\@\@) {

  my($listref) = shift;
  my(@molecule) = @{$_[0]}; shift;
  my(%started_torsions) = ();

  chdir($starting_directory) or
  append_to_log("Unable to change to $starting_directory in " .
          "start_needed_torsions(), this is a critical error, " .
          "exiting") and
     die "Critical error changing directories, see log for details";

  foreach (@$listref) {
  if ( $_ =~
     m/^Dir:\ ([^\ ]+)            # Qdb directory
        \ Parent\ bond\ (\d+)-(\d+): # Parent bond #'s
        \ Qdb\ bond\ (\d+)-(\d+):    # Qdb bond #'s
        \ qdb[^+-]+$                  # Rest of unmarked line
        /x                           # End pattern
     ) {
    my($dir) = $1;
    my($patom1) = $2;
    my($patom2) = $3;
    my($atom1) = $4;
    my($atom2) = $5;

    my $tenative_name = "$db_path/$dir/torsion_$atom2-$atom2";

    # Important last minute correction.  We do _not_ need to submit
    # torsions if one of the atoms is one of the monovalent atoms.
    # We need to check this before we run a torsion.  If it isn't a
    # sensible torsion, we simply mark the end of a line with a +.
    # There are other (odd) exceptions where this bug crops up as
    # well.  In particular, Carbonyl's are terminal as well, and
    # should not have torsions run on them.  They will be detected
    # as individual cases here, but this section should be rewritten
    # to check the connectivity of the fragment before asking for a
    # torsion.
    if ( $molecule[$patom1][0] eq "H" or
       $molecule[$patom2][0] eq "H" or
       $molecule[$patom1][0] eq "F" or
       $molecule[$patom2][0] eq "F" or
       $molecule[$patom1][0] eq "Cl" or
       $molecule[$patom2][0] eq "Cl" or
       $molecule[$patom1][0] eq "Br" or
       $molecule[$patom2][0] eq "Br" or
       $molecule[$patom1][0] eq "I" or
       $molecule[$patom2][0] eq "I" ) {
    # Don't submit this, but mark the list
    $_ .= "+";
    } else {
    unless (exists($started_torsions{"$tenative_name"})) {
      $_ .= "-";
      system("./torsion_driver.pl " .
         "$db_path/$dir $atom1 $atom2 $$ &");
      $started_torsions{"$tenative_name"} = 1;
    }
    }

    # Note:  The previous system call seems to be a bit slow (7 real
    # seconds for about 50 calls).  It _might_ be faster to implement
    # the call with a fork/exec combination.  I'm not fully
    # familiar with reaping, which is what needs to be done to the
    # child processes, and until I mess with it, the time it takes
    # to start the children will not be of concern.  If there are
    # eventually problems with too many processes being started
    # too quickly, we can simply prepare all of the requests, and
    # call a small program whose job it is to start the torsion
    # drivers over a period of time.  Note that not submitting
    # silly torsions speeds this up significantly *grin*.
  }
  }

  chdir("$db_path/control/qis") or
  append_to_log("Unable to change to $db_path/control/qis in " .
          "start_needed_torsions(), this is a critical error, " .
          "exiting") and
          die "Critical error changing directories, see log for details";

  return;
}

# The following function is pretty basic, but useful for re-creating
# the job file
sub print_map_list (@) {

  my(@list) = @_;

  print join("\n", @list) . "\n";

  return;
}

# The following function updates the job described in the first
# argument, overwriting it without abandon.  It's designed to be used
# as a sort of checkpoint save.  When called after initialization, it
# exactly reproduces the input file, with the exception of the
# stereochemical descriptors, which will be sorted
sub update_job($\@\@\@\@) {

  my($filename) = shift;
  my($p_ref) = shift;
  my($list1ref) = shift;
  my($list2ref) = shift;
  my($flistref) = shift;

  my($i);

  # No worries about this use statement being here, it's loaded at
  # compile time, and is global to this package
  use SelectSaver;

  open (TMP, ">input/$filename") or
  append_to_log("Unable to open input/$filename for writing in " .
          "update_job().  This is a critical file error, exiting") and
          die "Critical file error, see log for details";

  my($saver) = new SelectSaver("TMP");

  print "Begin parent molecule:\n";
  print_molecule(@$p_ref);
  print "End molecule output\n";

  print "Begin atom map list:\n";
  print_map_list(@$list1ref);
  print "End atom map list:\n";

  print "Begin bond map list:\n";
  print_map_list(@$list2ref);
  print "End bond map list:\n";

  print "Begin new fragments:\n";
  for $i ( 0 .. $#$flistref ) {
  print "Fragment list molecule number $i\n";
  print_molecule( @{ @{ $flistref }[$i] } );
  print "End molecule output\n";
  }

  print "Output complete\n";

  close(TMP);

  return;
}

# The following function simply returns a true or false depending on
# Whether the string could have come from get_CHNO_name or not.  Note
# that it will (before processing the string) discard any quantity
# of the string starting from the first "-", and going until the
# end.
sub is_CHNO_name ($) {

  my($string) = shift;

  ( $string, undef ) = split ("-", $string, 2);

  if ( $string =~
     m/^(            # Note the anchor
        (C([2-9]|[\d]*))?  # These match C9 or C13, but not C1
        (H([2-9]|[\d]*))?  # The ? means 0 or 1 matches (sorry,
        (N([2-9]|[\d]*))?  # it helps me *grin*).
        (O([2-9]|[\d]*))?
          |
        Other        # The whole funny pattern could also
      )$            # match simply 'other'
    /x) {            # Note the anchor on previous line
                  return 1;
                } else {
                  return 0;
                }
}

# The following function searches the fragment list given in the second
# argument for the atom found in the database directory given in the
# first argument.  Once it finds the correct fragment, it notes it's
# index in the fragment list, and searches in the <atom|bond>_map_list
# given in the third argument for lines that need to be changed from
# Fragment references to Qdb references, and makes the appropriate
# change
sub change_frag_to_qdb_in_list($\@\@) {

  my($dir_base_name) = shift;
  my($frag_list_ref) = shift;
  my($map_list_ref) = shift;
  my(@frag_list) = @$frag_list_ref;

  my($i);
  my($frag_index) = -1;

  my($dir) = "$db_path/$dir_base_name";
```

```
  # Find which fragment matches in the database
  for $i ( 0 .. $#$frag_list_ref ) {
  my(@this_mol) = @{@$frag_list_ref[$i]};
  unless ( is_molecule_unique($dir, @this_mol ) ) {
    # We found our molecule
    $frag_index = $i;
  }
  }

  if ($frag_index == -1) {
  append_to_log("Unable to find directory for fragment in " .
        "change_frag_to_qdb_in_list().  This indicates a " .
        "logical error on the part of the program, and needs " .
        "to be repaired.  Server shutting down");
  die "Critical logic error, see log for details";
  }

  # Ok, we know what fragment we're looking for, let's find the right
  # members of the map list, and correct them.

  foreach (@$map_list_ref) {
  if ( $_ =~
     /^(Parent .*:) Frag (.*:) frag_list offset ([\d]+) (.*)/ ) {
    if ( $3 == $frag_index ) {
    $_ = "Dir: $dir_base_name $1 Qdb $2 qdb $4";
    }
  }
  }

  # The final step is to remove this fragment from our fragment list,
  # since we've indicated it doesn't belong there.
  splice(@$frag_list_ref, $frag_index, 1);

  return;
}

# The following function simply removes a message from the messages
# meant for us.  It also removes the message from the global
# hash %completed_requests.  It has unfortunately not had extensive
# testing
sub remove_message ($) {

  my($message) = shift;

  my($is_open) = 0;
  open(TMP, "<$db_path/control/message.$$") and $is_open = 1;

  unless($is_open) {
  append_to_log("Unable to open <$db_path/control/message.$$ " .
        "for reading in remove_message().  We may have been " .
        "asked to remove a message when there was no messages? " .
        "this indicates a serious logical error, exiting");

  die "Critical logical error, see log for details";
  }

  flock(TMP, LOCK_EX);

  # We'll leave the file open and locked, and re-open it without
  # giving up our lock, this guarantees it won't change.

  my(@work_list) = <TMP>;

  my(%tmp_hash);
  foreach (@work_list) {
  $tmp_hash{$_} = 1;
  }

  delete($tmp_hash{$message});
  delete($completed_requests{message});

  open(TMP, ">$db_path/control/message.$$") or
  append_to_log("Unable to open $db_path/control/message.$$ " .
        "for writing in remove_message().  This is a critical " .
        "file error, exiting") and
    die "Critical logical error, see log for details";

  print TMP join("\n", keys(%tmp_hash)) . "\n";

  # Release the lock and go home
  flock(TMP, LOCK_UN);
  close(TMP);

  return;
}

# The following function moves a directory.  The directory must be
# flat (i.e., it must not have any subdirectories)
sub move_flat_directory ($$) {

  my($source) = shift;
  my($dest) = shift;

  my(@file_list) = glob("$source/*");

  # Launder the file list
  foreach (@file_list) {
  ($_) = $_ =~ m%([\w/.-]+)%;  # %'s used since / is in the pattern.
  }

  # Get the basename of the directory
  my(@work_list) = split("/", $source);
  my($basename) = pop(@work_list);
  my($original_directory) = join("/", @work_list);

  # Create the new directory
  mkdir($dest, 0775) or
  append_to_log("Unable to create directory $dest in " .
        "move_flat_directory().  This is a critical file error, " .
        "daemon exiting") and
        die "Critical file error, see log for details";

  # Write our is_hosed file to indicate we're not done
  open(TMP, ">$dest/is_hosed") or
  append_to_log("Unable to create is_hosed in destination directory" .
        "in move_flat_directory().  This is a critical file " .
        "error, daemon exiting") and
    die "Critical file error, see log for details";
  close(TMP);

  # Okies, the directories are done, now we just link and unlink to
  # our heart's content.
  foreach (@file_list) {
  @work_list = split("/", $_);
  my($filename) = pop(@work_list);

  link("$_", "$dest/$filename") or
    append_to_log("Unable to create new link for $dest/$filename, " .
        "this is a critical error, exiting")
    and die "Critical file error, see log for details";

  unlink("$_") or
    append_to_log("Unable to unlink $_, this is a critical error, " .
        "exiting") and
    die "Critical file error, see log for details";
  }

  # Remove the original directory
  rmdir($original_directory) or
  append_to_log("Unable to delete original directory " .
        "$original_directory, this is a critical error, exiting")
    and die "Critical file error, see log for details";

  # And finally, we can remove is_hosed
  unlink("$dest/is_hosed") or
  append_to_log("Unable to unlink $dest/is_hosed, this is a " .
        "critical error, exiting") and
    die "Critical file error, see log for details";

  return;

}

# The following function builds an empty and 'ready to go' scaffold
# directory for initial fragment optimization.  It builds the master
# directory in in_progress with the same name as the job in the
# input directory.  Given this structure, it's _critical_ to not
# overwrite jobs in the input directory.  This task will presumably
# be done by a master program that runs qdb_check ... so it won't be
# a problem, but when entering jobs manually, the user must be
# aware of this fact.
sub build_submit_scaffold_dir($\@) {
  my($job_name) = shift;
  my($molref) = shift;
  my($i) = 0;
  my(@mol) = @$molref;
  my($is_open) = 0;
  my(@work_list);
  my(@other_names);

  # Prepare a list for submission to get_CHNO_formula
  my(@lab) = ();
  foreach (@mol) {
  push(@lab, $_->[0]);
  }

  # The following function updates %qdb_entries to be current
  get_database_entries();

  # And add any new directories that may also be in progress
  open(TMP, "<in_progress/qdb_entries_in_progress") and $is_open = 1;
  if ($is_open) {
  @work_list = <TMP>;
  chomp(@work_list);
  close(TMP);
  foreach (@work_list) {
    # Split off the second value, we only want the first
    ( my($val), undef ) = split(/ /, $_, 2);
    push(@other_names, $val);
  }
  }        # If it's not open, there must be nothing to add *grin*

  # We now create the directory in in_progress, under $job_name.  It
  # should be complete before we allow ourselves to move on.  Once it's
  # complete, we need to add any .com files, etc.  We can then submit
  # the gaussian jobs to the que.  Note that if the jobs are already
  # complete (because we're recovering from a crash, or whatnot), it's
  # not a problem, since the qdb_local_submit.pl daemon will not
  # re_submit finished jobs, it'll just tell us it's finished.  Don't
  # forget to write all entries, including stereochemical descriptors,
  # and frozen bonds.

  # We will simply assume that all entries given to us are unique, and
  # we won't check again.  We rely on a previous call to
  # is_molecule_unique() for this purpose.

  my($new_qdb_entry_name) =
  get_unique_name_in_dir( get_CHNO_formula(@lab),
            "$db_path/control/qis/in_progress", 1,
            @other_names);

  my($working_dir) = "$db_path/control/qis/in_progress/$job_name";

  mkdir("$working_dir", 0775) or
  append_to_log("Unable to create directory " .
        "$working_dir with proper permissions in " .
        "build_submit_scaffold_dir().  This is " .
        "a critical error, exiting") and
    die "Critical file error in build_submit_scaffold_dir(), " .
      "see log for details";

  $working_dir .= "/$new_qdb_entry_name";

  mkdir("$working_dir", 0775) or
  append_to_log("Unable to create directory " .
        "$working_dir with proper permissions in " .
        "build_submit_scaffold_dir().  " .
```

```
        "This is a critical error, exiting") and
        die "Critical file error in build_submit_scaffold_dir()";

# And start adding the properly formatted files

# Create Original_structure.raw
open(TMP, ">$working_dir/Original_structure.raw") or
append_to_log("Unable to open $working_dir/Original_structure.raw " .
        "for writing in build_submit_scaffold_dir().  This is a " .
        "critical file error, and indicates a major problem, " .
        "exiting") and
  die "Critical file error, see log for details";

# Note that gaussian 98 _requires_ that the input for coordinates
# is formatted as a float, not just an integer.  This means we'll
# need to use printf to print the values of the coordinates.
foreach (@mol) {
  print TMP $_->[0] . ",";
  # This was the old line join(",", @{$_->[1]} ) . "\n";
  printf("%.8f,%.8f,.8%f\n",
    @{$_->[1]}->[0], @{$_->[1]}->[1], @{$_->[1]}->[2]);
}
close(TMP);

# Create Connectivity.raw
open(TMP, ">$working_dir/Connectivity.raw") or
append_to_log("Unable to open $working_dir/Connectivity.raw " .
        "for writing in build_submit_scaffold_dir().  This is a " .
        "critical file error, and indicates a major problem, " .
        "exiting") and
  die "Critical file error, see log for details";

print TMP join("\n", @{$mol[0][2]}) . "\n";
close(TMP);

# Create Qcodes
open(TMP, ">$working_dir/Qcodes") or
append_to_log("Unable to open $working_dir/Qcodes " .
        "for writing in build_submit_scaffold_dir().  This is a " .
        "critical file error, and indicates a major problem, " .
        "exiting") and
  die "Critical file error, see log for details";

foreach (@mol) {
print TMP join("\n", @{$_->[3]} ) . "\n";
}
close(TMP);

# The following section determines if a Stereochemical_descriptors file
# must be written, and does so if it does, after sorting the results in
# numerical order
my($needs_sdescrip) = 0;
$i = 0;
@work_list = ();
my(%work_hash);
foreach (@mol) {
if ( $_->[4] ) {
  $needs_sdescrip = 1; $work_hash{$i} = $_->[4];
}
$i++;
}
if ( $needs_sdescrip ) {
open(TMP, ">$working_dir/Stereochemical_descriptors") or
  append_to_log("Unable to open $working_dir/" .
        "Stereochemical_descriptors " .
        "for writing in build_submit_scaffold_dir().  This is a " .
        "critical file error, and indicates a major problem, " .
        "exiting") and
  die "Critical file error, see log for details";
@work_list = sort { $a <=> $b } keys(%work_hash);
foreach (@work_list) {
  print TMP "$_ $work_hash{$_}\n";
}
close(TMP);
}

# The following section prints the contents of Frozen_bonds (This
# can't be done until a Original_structure.raw file is in place)
@work_list = ();
my($work_string) = "$starting_directory/qdb_maintenance_utilities/" .
"utilities/determine_frozen_bonds $working_dir";
open(TMP, "$work_string |") or
append_to_log("Unable to run $work_string in " .
        "build_submit_scaffold_dir(), this is a critical " .
        "error, exiting")
  and die "Critical error running external file";
@work_list = <TMP>;
$work_string = join("", @work_list);
close(TMP);
# Open target file ... and
if ($?) {       # In otherwords, the called program did something
open(TMP, ">$working_dir/Frozen_bonds") or
  append_to_log("Unable to open $working_dir/Frozen_bonds for " .
        "writing in build_submit_scaffold_dir().  This is a " .
        "critical file error, exiting") and
  die "Critical file error, see log for details";
print TMP $work_string;
close(TMP);
}

# And finally, the follwing section creates the Initial_optimization.com
# file.  Remember that because format_for_g98.pl was shoddily written
# (by myself, admittedly), we need to change directories to there
# before we run it, and change back here after.
chdir($starting_directory);
$work_string = "./format_for_${ab_initio_program}.pl " .
"$working_dir Original_structure.raw Initial_optimization";
system($work_string);
chdir("$db_path/control/qis") or
append_to_log("Unable to change back to $db_path/control/qis " .
        "in build_submit_scaffold_dir(), this is a critical " .
        "error, exiting") and
  die "Unable to change to input server's home directory " .
    "($db_path/control/qis)";

  # The scaffold directory is complete!  The last and final task is for
  # the function to put the request on the que.  Since the que is a
  # 'public' file, we need to do correct file locking semantics when
  # using it

  my(%que) = ();
  $is_open = 0;
  open(QUE, "<$db_path/control/que") and $is_open = 1;

  if ($is_open) {
  @work_list = <QUE>;
  close(QUE);
  chomp(@work_list);

  # Since only the 3rd item in the que is important, we will
  # make another list with only those
  for (@work_list) {
    ( undef, undef, undef, $_ ) = split(",", $_);
  }

  # Hash it
  for (@work_list) {
    if ( $_ ) {
    $que{$_} = 1;
    }
  }
  close(QUE);

  } else {
  # Make an empty que
  $que{""} = 0;
  }

  # The hash now necessarily has no repeats, and can be used to prevent
  # adding duplicates

  open(QUE, ">>$db_path/control/que") or
  append_to_log("Unable to open $db_path/control/que for " .
        "appending in build_submit_scaffold_dir().  This is " .
        "a critical file error, exiting") and
    die "Critical file error, see log for details";
  flock(QUE, LOCK_EX);
  seek(QUE, 0, 2);
  $work_string = "$working_dir/Initial_optimization." .
             "$ab_initio_suffix";
  unless (defined($que{$work_string})) {
  print QUE "$this_user,$this_host,$$,$work_string\n";
  }

  flock(QUE, LOCK_UN);
  close(QUE);

  # Before we can leave, we need to enter our information in the
  # qdb_entries_in_progress file.  This file is needed to prevent
  # duplicate scaffold directories.
  $is_open = 0;
  open(TMP, "<in_progress/qdb_entries_in_progress") and $is_open = 1;

  if ($is_open) {
  # It's open.  Do our work and close it.  We'll check for duplicate
  # names, since a duplicate entry here would indicate a logical
  # error in the program, and we'd rather die than be imperfect
  # *grin*
  @work_list = <TMP>;
  close(TMP);
  push(@work_list, $new_qdb_entry_name);
  %work_hash = ();
  foreach ( @work_list ) {
    my($val);
    ( $val, undef ) = split(" ", $_);
    if ( exists($work_hash{$val}) ) {
    # We have our error condition
    append_to_log("Duplicate entry in file in_progress/" .
            "qdb_entries_in_progress detected in function " .
            "build_submit_scaffold_dir().  This is a logical " .
            "error, please contact development to correct it");
      die "Critical logical error encountered.  See log for " .
      "details";
    }
    $work_hash{$val} = 1;;
  }
  }

  # Now, simply put our new entry into the file.
  open(TMP, ">>in_progress/qdb_entries_in_progress") or
  append_to_log("Unable to open input/qdb_entries_in_progress " .
        "for appending in build_submit_scaffold_dir().  This is " .
        "a critical file error, exiting") and
    die "Critical file error, see log for details";
  print TMP "$new_qdb_entry_name " .
        "$db_path/control/qis/in_progress/$job_name/\n";
  close(TMP);

  return;

}

# This little function takes an original name, and a target directory,
# and returns some version of the original name which would be new
# to the given directory.  If the original name is not available, it
# simply returns the original name with "-<number>" appended, where
# <number> is the smallest suitable integer.  The third argument is
# optional, and if provided and it evaluates to true, it will
# force the program to never return simply the base name, but will
# always append at least a -0 to the name before returning.  The final
# optional list will contain additional directory names that should
# be considered unavailable
sub get_unique_name_in_dir ($$;$@) {

  my($original_name) = shift;
  my($target_directory) = shift;
  my($extra_option) = shift;
  my(@other_dirs) = @_;
```

```
  my(@shrunk_dir_list) = ();
  my(@suffix_list);
  my($i);

  opendir(TMP_DIR, $target_directory) or
  append_to_log("Unable to open $target_directory " .
          "for a directory reading in get_unique_name_in_dir, exiting")
    and die "Critical file error, see log for details";
  my(@dir_list) = readdir(TMP_DIR);
  closedir(TMP_DIR);

  # Add other directories to the list of names to not duplicate
  if ($other_dirs[0]) {
  push(@dir_list, @other_dirs);
  }

  # And, eliminate any "non-competing" names
  @shrunk_dir_list = grep(/^$original_name/, @dir_list);

  # We need to note if the true base name exists, and remove it from
  # the list if necessary
  my($base_exists) = 0;
  for ( $i = 0; $i <=$#shrunk_dir_list; $i++) {
  if ( $shrunk_dir_list[$i] eq $original_name ) {
    $base_exists = 1;
    # And remove it from this list
    splice(@shrunk_dir_list, $i, 1);
    $i--;
  }
  }

  # If there are no competing names, we can simply return the original,
  # as long as this is allowed by the optional argument;
  if ($extra_option) {
  unless (@shrunk_dir_list) {
    @shrunk_dir_list = ( "$original_name-1" );
  }
  } else {
  # If the base name wasn't in the requested directory, and we
  # didn't specify the optional flag, just return the base name.
  if ( !$base_exists ) {
    return $original_name;
  }

  # And in this case, we don't need to test for the @shrunk_dir_list
  # being empty, since the base wouldn't have existed if it was
  # empty

  }

  # Finally, shrink it down to just the suffixes
  foreach (@shrunk_dir_list) {
  ( undef, my($val) ) = split(/-/, $_);
  push(@suffix_list, $val);
  }

  # And finally, sort it
  @suffix_list = sort {$a <=> $b} @suffix_list;

  $i = 0;
  # Now, look for the new name.
  while ( @suffix_list && $i == shift @suffix_list ) {
  $i++;
  }

  # Finally, return our new name
  return $original_name . "-$i";

  return;
}

# The following function takes as arguments a directory name, and a
# molecule.  It then look through all of the directories with the same
# CHNO formula and compares the qcodes of the fragment and the
# molecule in that directory.  If every qcode is not an exact match,
# 0 is returned, if it is the same, 1 is returned.  I don't have a proof
# that different molecules can have the same set of qcoes (unordered),
# but it seems unlikely.  Perhaps Edgardo has something on this?
# Note that in a later edition, it will handle for a directory name
# either a single directory to search in, or a parent directory, with
# several entries in it.  It will determine this by calling is_CHNO_name()
# on the last portion of $dir.  If it's a single directory search
# it does just that.
sub is_molecule_unique ($@) {

  my($dir) = shift;
  my(@mol) = @_;

  my($line);
  my($is_match) = 1;
  my($is_single_directory_search) = 0;
  my(@work_list);
  my($basename);
  my($is_open) = 0;

  # Prepare a list for submission to get_CHNO_formula
  my(@lab) = ();
  foreach (@mol) {
  push(@lab, $_->[0]);
  }

  my($chno_name) = get_CHNO_formula(@lab);

  # Determine if we're doing a single directory search, or a mass
  # directory search.
  @work_list = split("/", $dir);
  ( $basename, undef ) = split("-", $work_list[$#work_list]);
  if ( is_CHNO_name($basename) ) {
  # We're doing a single directory search
  $is_single_directory_search = 1;
  }

  my(%qcodes_under_scrutiny) = ();

  foreach (@mol) {
  $qcodes_under_scrutiny{join(" ", @{$_->[3]})} = 1;
  }

  if ( $is_single_directory_search ) {
  # A single directory search is much easier

  $is_match = 1;

  if ( $basename ne $chno_name ) {
    return 1;
  }
  $is_open = 0;
  open(TMP, "$dir/Qcodes") and $is_open = 1;

  if ( $is_open ) {
    while ( $line = <TMP> ) {
    chomp($line);
    unless (exists( $qcodes_under_scrutiny{$line}) ) {
      $is_match = 0;
      close(TMP);
      last;
    }
    }
    close(TMP);
  }

  return 0 if ($is_match);
  return 1;
  } else {
  # If we can't open the directory we've been given, it's clearly
  # not a match.
  my(%work_hash) = ();
  opendir(DB_DIR, $dir) or return 1;

  @work_list = readdir(DB_DIR);
  closedir(DB_DIR);

  # And ... hash it
  for (@work_list) {
    $work_hash{$_} = 1;
  }

  my(@target_directories) = grep(/^$chno_name/, keys(%work_hash));

  foreach (@target_directories) {
    $is_match = 1;
    open(TMP, "$dir/$_/Qcodes") and $is_open = 1;
    if ( $is_open ) {
    while ( $line = <TMP> ) {
      chomp($line);
      unless (exists( $qcodes_under_scrutiny{$line}) ) {
      $is_match = 0;
      close(TMP);
      last;
      }
    }
    close(TMP);
    }

    return 0 if ($is_match);
  }
  return 1;
  }
}

# The following function simply reads the first list and returns it's
# members verbatim from the provided list;
sub read_qdb_input_list(\@) {

  my($list_ref) = shift(@_);
  my(@return_list) = ();
  my($line);

  while ( @$list_ref and
    ( $line = shift(@$list_ref) ) !~ /^Begin ([\w]+) map list:$/ ) {
  }

  return undef unless($line);

  $line =~ /^Begin ([\w]+) map list:$/;
  # Read until the end
  while ( @$list_ref and
      ($line = shift(@$list_ref) ) !~ /^End $1 map list:$/ ) {
  push(@return_list, $line);
  }

  return undef unless($line);

  return @return_list ? @return_list: undef;
}

# The following function prints out all of the information contained
# in a molecule.  It's mainly here for demonstration purposes
sub print_molecule (@) {

  my(@molecule) = @_;

  print "Begin coordinates\n";
  foreach (@molecule) {
  print $_->[0];
  print ",";
  print join(",", @{$_->[1]} ) . "\n";
  }

  print "Begin connectivity\n";
  print join("\n", @{$molecule[0][2]}) . "\n";

  print "Begin qcodes\n";
  foreach (@molecule) {
  print join("\n", @{$_->[3]} ) . "\n";
  }

  my($needs_sdescrip) = 0;
```

```
  my($i) = 0;
  my(@work_list) = ();
  my(%work_hash);
  foreach (@molecule) {
  if ( $_->[4] ) {
    $needs_sdescrip = 1; $work_hash{$i} = $_->[4];
  }
  $i++;
  }
  if ( $needs_sdescrip ) {
  @work_list = sort { $a <=> $b } keys(%work_hash);
  print "Begin stereochemical descriptors\n";
  foreach (@work_list) {
    print "$_ $work_hash{$_}\n";
  }
  }
  return;
}

# The following subroutine reads the next molecule in the given array
# into a (somewhat) complex data structure.  The structure itself is
# a simple array, with the indexes corresponding to the atom number.
# The actual structure of the array is as follows:
# @mol        The entire molecule
# $mol[$num]    A reference to information on atom $num
# $mol[$num][0]    The label of the atom at $num
# $mol[$num][1]    A reference to the coordinates of the atom at $num
# $mol[$num][1][0]  The x coordinate of the atom at $num
# $mol[$num][1][1]  The y coordinate of the atom at $num
# $mol[$num][1][2]  The z coordinate of the atom at $num
# $mol[0][2]    A reference to the connectivity list for the molecule
#          Note:  Any value besides 0 will be invalid!
# $mol[$num][3]    The list of qcodes for that atom.
# $mol[$num][4]     A value of the stereochemical descriptor, if that
#          atom actually has one
# Note:  The caller will get an error if they try to call this function
# (or any function that takes a reference) with an undef reference

sub read_molecule (\@) {

  my($mol_ref) = shift(@_);
  my($line);
  my($i);
  my(@return_array);
  my(@work_list);

  # If at any point in this routine we run out of list to read, simply
  # return undef

  # Read lines until we get to the start of coordinates
  while ( $line = shift(@$mol_ref) and
    $line !~ /^Begin coordinates$/ ) {
  }

  return undef unless($line);

  $i = 0;
  while ( @$mol_ref and
      ($line = shift(@$mol_ref)) !~ /^Begin connectivity$/ ) {
  @work_list = split(/,/, $line);
  $return_array[$i][0] = shift(@work_list);
  $return_array[$i][1] = [ @work_list ];
  $i++;
  }

  return undef unless($line);

  # We're at the connectivity section, let's read it.

  @work_list = ();
  while ( @$mol_ref and
      ($line = shift(@$mol_ref)) !~ /^Begin qcodes$/ ) {
  push(@work_list, $line);
  }
  $return_array[0][2] = [ @work_list ];

  return undef unless($line);

  $i = 0;
  # Done with the connectivity, let's read the qcodes
  while ( @$mol_ref and
      ( $line = shift(@$mol_ref ) ) !~
      /^Begin stereochemical descriptors$/ and
      $line !~ /^End molecule output$/ ) {
  push( @{$return_array[$i][3]}, $line );
  $i++;
  }

  return undef unless($line);

  # And finally, we can look for stereochemical descriptors
  if ( $line =~ /^Begin stereochemical descriptors/ ) {
  while ( @$mol_ref and
      ( $line = shift(@$mol_ref) ) !~ /^End molecule output$/ ) {
    ($i, undef) = split(/ /, $line);
    (undef, $return_array[$i][4]) = split(/ /, $line);
  }
  }

  return undef unless($line);

  # That's it, we're done, return this array

  return @return_array;
}

sub get_database_entries {

  %qdb_entries = ();

  opendir(DB_DIR, $db_path) or die
  "Unable to open $db_path for a directory list in " .
    "get_database_entries(), exiting\n";

  my(@work_list) = readdir(DB_DIR);

  # And ... hash it
  for (@work_list) {
  $qdb_entries{$_} = 1;
  }

  closedir(DB_DIR);

  # Before we return, remove the extraneous entries
  delete ( $qdb_entries{"."} );
  delete ( $qdb_entries{".."} );
  delete ( $qdb_entries{"control"} );

  return;
}


# The following function does exactly as implied by its name.
sub append_to_log (@) {
  my($message) = join(" ", @_);
  my($right_now);

  # Open the log file
  open (MY_LOG, ">>$db_path/control/qdb_input_server.log") or die
  "Cannot open server log for writing in append_to_log ... " .
    "exiting";

  $right_now = localtime;
  print MY_LOG "$right_now\n";
  print MY_LOG "$message\n";
  close(MY_LOG);
  return;
}

# The following function looks through the array passed to it and
# the CwHxNyOz string for it's molecular formula.  If there are none
# of those atoms in the molecular formula, it returns "Other"
sub get_CHNO_formula (\@) {
  my($labref) = shift;
  my(@labels) = @$labref;
  my($C) = 0;
  my($H) = 0;
  my($N) = 0;
  my($O) = 0;
  my($label) = "";
  my($line);

  foreach $label (@labels) {

  $label = uc($label);

  if ( $label eq "C" ) {
    $C++;
  } elsif ( $label eq "H" ) {
    $H++;
  } elsif ( $label eq "N" ) {
    $N++;
  } elsif ( $label eq "O" ) {
    $O++;
  }
  }

  $label = "";
  if ( $C != 0 ) {
  $label .= "C";
  if ($C >= 2 ) {
    $label .= $C;
  }
  }
  if ( $H != 0 ) {
  $label .= "H";
  if ($H >= 2 ) {
    $label .= $H;
  }
  }
  if ( $N != 0 ) {
  $label .= "N";
  if ($N >= 2 ) {
    $label .= $N;
  }
  }
  if ( $O != 0 ) {
  $label .= "O";
  if ($O >= 2 ) {
    $label .= $O;
  }
  }
  if ( $label eq "" ) {
  return "Other";
  }
  return $label;
}

# The following section contains the program's signal handlers

# This handler lets the program know if it got an alarm
sub wake_up {
  $got_alarm = 1;
  return;
}

# The following handler ignores SIGHUP, which shells send upon exit.  It
# appears to have the side effect of setting off the alarm.
sub handle_hup {
  return;
}

# This handler allows the program to exit gracefully, though it's not
# likely to be used often, since it is a daemon, after all
sub quit_now {
  $loop_forever = 0;
  $got_alarm = 0;
  return;
```

```
}
```

Force field creation programs

qdb_check.h

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* Includes */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include "fraggen.h" /* Fraggen contains atom.h, which contains vector.h */
#include "../general/my_socket.h"
#include "qdb_shared_functions.h"

/* Conditional Includes */
#ifndef ATOM_H
#define ATOM_H
#include "../general/atom.h"
#endif

#ifndef VECTOR_H
#define VECTOR_H
#include "../general/vector.h"
#endif

/* The following is a memory debugging library */
#ifdef DMALLOC
#include <dmalloc.h>
#endif

/* Defines */

/* The following gives extra output (currently sent to stderr).  What is */
/* implemented now is a simple bar that tells how close we are to being */
/* done with our database queries.  Comment it out, or disable it from */
/* the makefile if this is not desired */
#define EXTRA_OUTPUT

#ifndef MAXSTR
#define MAXSTR 256
#endif

#ifndef QDEPTH
#define QDEPTH 20 /* Important!  This is also defined in assign_qcodes.c, */
          /* which should _not_ be dependant on the qdb code, */
          /* (which is why this header isn't included in it) */
#endif

#ifdef __unix__
/* Put UNIX specific defines here.  Note that directory access will */
/* definately differ from system to system */
#endif

/* Macros */

#define BLANK_QDB_RESULT(var)        \
strcpy(var.directory, "");           \
var.atom1 = var.atom2 = -1;          \
var.s_atom1 = var.s_atom2 = -1;        \
var.tolerance1 = var.tolerance2 = -1.0;    \
var.offset = -1;

/* Typedefs and structs */

#ifndef BOOLEAN
#define BOOLEAN
typedef enum {false = 0, no = 0, true = 1, yes = 1} boolean;
#endif

typedef struct dihed {
  int list_index;
  int atom1_offset;
  int atom2_offset;
} dihed;

typedef struct qdb_result {
  char directory[MAXSTR];
  int atom1;
  int atom2;
  float tolerance1;
  float tolerance2;
  int s_atom1; /* <--- That's source atom1 */
  int s_atom2;
  int offset;
} qdb_result;

/* Function prototypes */

  /* In qdb_check_functions.c */
void Usage();
void error_exit(char *message);
void warn_out(char* message);
void chk_mem(void);
boolean is_qcode_match(atom *first_atom, atom *second_atom,
  float tolerance);
dihed *add_dihedral(dihed* dihedral_list, atom **molecule_list,
  atom *atom1, atom *atom2);
atom **add_to_molecule_list(atom *some_atom, atom **list);
qdb_result is_in_qdb(atom *molecule_base, atom *second_atom, float tolerance,
  int socket_handle);
char *qdb_special_query(atom *atom1, atom* atom2, int socket_handle,
  int op_code, void *other );
dihed is_qcode_match_in_molecule_list(atom *original, atom* second_atom,
  atom **list, float tolerance);
qdb_result **add_dbresult_to_qdb_result_list(qdb_result **original_list,
  int flag, qdb_result my_result);
float reduce_tolerance(float tolerance, int range);
void print_qdb_result(qdb_result result);
void print_result_list(FILE *destination, qdb_result **list);

/* The following are functions in qdb_check_functions.c that are */
/* moved out of main() to keep it less cluttered, and more readable */
char *prepare_asymmetry_query(qdb_result work_result, atom *center,
  float tolerance, int asymmetry_range);
char *get_rcfile_resource(FILE *rc_file, const char* id);
atom *read_init_formatted_coordinates(FILE *read_stream);
void read_formatted_connectivity(FILE *read_stream, atom *member);
int compare_asymmetry_environment_atom(qdb_result result,
  atom *parent_molecule, atom **compare_list, int range, float tolerance);
int compare_asymmetry_environment_db(qdb_result result,
  atom *parent_molecule, int socket_handle, int range, float tolerance);

  /* in ../log2str/get_bond_order.c */
float get_bond_order(int atom1, int atom2, float distance);

  /* in assign_qcodes.c */
int assign_qcodes(atom *some_atom);

  /* in ../general/my_socket.c */
/* int get_new_socket(const char* hostname, const unsigned int port); */
/* boolean sendall(int socket_descriptor, char *to_send, int *length); */
```

fraggen.h

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* Includes */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "qdb_shared_functions.h"

/* Conditional Includes */
#ifndef ATOM_H
#define ATOM_H
#include "../general/atom.h"
#endif

#ifndef VECTOR_H
#define VECTOR_H
#include "../general/vector.h"
#endif

/* The following is a memory debugging library */
#ifdef DMALLOC
#include <dmalloc.h>
#endif

/* Defines */
```

```
#define H_BIT 30

/* Typedefs and structs */

/* Function prototypes */

atom *generate_fragment(atom *first_atom, atom *second_atom, int depth);
boolean isin_group(atom *this_atom, int bond_index);
void recurse_generate_fragment(atom *last_atom, atom *this_atom,
  int current_depth);
void wrap_recurse_generate_fragment(atom *this_atom, int target_depth);
boolean ami_new_visitor(atom *this_atom, int bond_number);
long int *mark_recurse_fragment ( atom* center, int depth );
void mark_recurse_fragment_core ( atom *center, atom* source, int depth );

/* Function prototype from assign_qcodes.c */
int assign_qcodes(atom *some_atom);

/* Function prototypes from qdb_check_functions.c */
atom **add_to_molecule_list(atom *some_atom, atom **list);
```

qdb_shared_functions.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include "qdb_shared_functions.h"

/* This is another standard function, placed in a 'standard' place */
void error_exit(char *message) {
  printf("%s ... exiting.\n", message);
  exit(0);
}

/* This is a standard small warning function.  It's in the shared */
/* functions because get_qcode_deviance() calls it */
void warn_out(char *message) {
  printf("Warning:  %s\n", message);

  return;
}

/* This function takes two vectors and a desired tolerance, and compares */
/* them.  It returns a float which is the deviance, a custom comparison */
/* function that can be changed as needed */

float get_qcode_deviance(long double *qcode1, long double *qcode2,
  int length) {

const long double slop = 0.000000000001;
int i, exact_match;
float weighting_factor, fractional_match;
double sum, diff, temp_double;

/* Error checking */
if (qcode1 == NULL) {
  warn_out("first atom passed to is_qcode match is NULL, this was most "
       "likely unintended, but should not be fatal");
}
if (qcode2 == NULL) {
  warn_out("second atom passed to is_qcode match is NULL, this was most "
       "likely unintended, but should not be fatal");
}

/* Test for exact part.  If the qcodes were generated on the same machine, */
/* they will be exact, otherwise, the slop may need to be adjusted */
for ( i = 0; i < length &&
  fabs(qcode1[i] - qcode2[i]) < slop; i++ ) {
}

exact_match = i;

/* And now we test the fractional part */
/* Note that this is a weighted squares.  The most important terms of the */
/* qcode vector will be the first few, and the terms near the end are */
/* ultimately of much less importance.  Therefore, we use an exponential */
/* weighting to emphasize the first terms more */
weighting_factor = 0.5;
sum = 0;
for ( i = exact_match; i < length; i++ ) {
  diff = exp( -1.0 * weighting_factor * (i - exact_match + 1) ) *
    fabs(qcode1[i] - qcode2[i]);
  sum += diff * diff;
}

/* And we need an _average_ squares difference, since this may be done */
/* with different values of length, and we want them to be comparable */
sum /= length - exact_match;
/* if sum is currently 0, we don't want to take the log of it */
/* Note here that we're defining the value 0.999 to be the 'practical */
/* perfect' match.  This could be checked for later if the calling */
/* routine cares */
if (sum == 0.0 ) {return ((float)exact_match + 0.999) ;}
temp_double = log(sqrt(sum));
sum = temp_double < 0 ? -1 * temp_double : 0;

/* Recall that fractional_match checks the match of ites beyond what we */
/* demanded an exact match for.  Empirically, if sum = 6 it is a very bad */
/* match, 10 is really quite good, 15 is excellent, 20 is nearly perfect, */
/* and 25 is a practical maximum.  As a result, to make the input more */
/* 'intuitive' for users of the program, the number they provide after the */
/* decimal point will be in the form of a percent, and the sum just */
/* calculated will be multiplied by four.  For translation then, 20 is a */
/* totally crappy match, while 40% is quite good, and 60% is about perfect */

fractional_match = sum >= 25 ? 0.999 : sum * 4 / 100;

return (float)exact_match + fractional_match;

}
```

qdb_check_functions.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include "qdb_check.h"
#include <stdio.h>
#include <string.h>

/* This file contains all of the functions needed by qdb_check.c that are */
/* not of general interest. */

void Usage(void) {

  printf("%s",
     "Usage:\n"
     "     qdb_check\n\n"
     "     Note that you _must_ have a valid .qdb_checkrc file with the \n"
     "     proper path name in it.  The details of the location of this \n"
     "     file will be looked at later\n\n"
     );

  exit(0);
}

void chk_mem(void) {

  system("ps -l|grep qdb_check|awk '{print $11}'");
  /* system("sleep 1"); */

  return;
}

/* The following function recurses (through all of the bonds) to each atom */
/* in the molecule exactly once.  target_state must be set to a value */
/* that is not in any of the atoms yet, or that atom will be skipped, */
/* (possibly skipping atoms past it as well).  */
void atom_recurse_demo(atom *source_atom, atom *this_atom, int target_state) {

  int i;

  /* Do anything that needs to be done on the first call here */
  /* This has been visited, so set the state */
  this_atom->state = target_state;

  if (source_atom == NULL) {
  /* Do first call actions here */
  }

  /* If there are any actions that should be performed on each atom visited */
  /* they can be inserted here */

  for (i = 0; i < this_atom->valence; i++) {
  /* SHOW_BONDS <--- label */
  if (this_atom->bond[i] != source_atom) {
    /* We can continue:  Check if the next atom has been visited */
    if ( this_atom->bond[i]->state != target_state ) {
    /* Then go on with it (call recursively) */
    atom_recurse_demo(this_atom, this_atom->bond[i], target_state);
    }
  }
  }

  return;
}

/* I have decided to break is_qcode_match() into two parts.  Sometimes, */
```

```
/* we may want not want to know just whether or not we've found the */
/* match, but also how good of a match it is.  For backwards */
/* compatability, is_qcode_match() will call the new function */
/* get_qcode_deviance() (which has a similar prototype), and simply */
/* return the boolean value as before */
boolean is_qcode_match(atom *first_atom, atom *second_atom,
             float tolerance) {

  float deviance;
  int exact_match;

  /* Error checking */
  if (first_atom == NULL) {
  warn_out("first atom passed to is_qcode_match is NULL, this was most "
       "likely unintended, but should not be fatal");
  }
  if (second_atom == NULL) {
  warn_out("second atom passed to is_qcode_match is NULL, this was most "
       "likely unintended, but should not be fatal");
  }

  exact_match = (int)(tolerance);

  /* error checking */
  if (exact_match > QDEPTH - 1 ) {
  warn_out("too high of a precision passed to is_qcode_match(), "
       "returning no");
  return no;
  }

  deviance = get_qcode_deviance(first_atom->qcode, second_atom->qcode,
               QDEPTH);

  if (deviance >= tolerance) { return yes; }

  /* If anything falls through the tests, return no */
  return no;

}

/* The following function takes an atom to check for, a list of molecules */
/* to check in, (the list should be null terminated), and the tolerance as */
/* arguments, and looks for a qcode match in the given list, returning */
/* the index of the list that matches.  The variable second_atom exists to */
/* check for membership of bond centered fragments.  If it is NULL, it's */
/* ignored.  If there is no match, it returns a dihedral with all elements */
/* = -1 */
dihed is_qcode_match_in_molecule_list(atom *original, atom *second_atom,
                   atom **list, float tolerance) {

  int i, j;
  atom *work_atom;
  dihed work_dihedral;

  /* Assign default values to work_dihedral */
  work_dihedral.list_index = -1;
  work_dihedral.atom1_offset = -1;
  work_dihedral.atom2_offset = -1;

  /* Error checking */
  if (list == NULL) {
  /* Return nothing, this is an empty list */
  return work_dihedral;
  }
  if (original == NULL) {
  warn_out("NULL atom passed to "
       "is_qcode_match_in_list().  This may be fatal");
  return work_dihedral;
  }

  for ( i = 0; list[i] != NULL; i++ ) {
  work_atom = list[i] - get_atom_offset(list[i]);

  /* Now, loop over all of the atoms in work_atom, looking for a match */
  while (work_atom != NULL) {
    if ( is_qcode_match(work_atom, original, tolerance) ) {
    if (second_atom == NULL) {
      work_dihedral.list_index = i;
      work_dihedral.atom1_offset = get_atom_offset(work_atom);
      work_dihedral.atom2_offset = -1;
      return work_dihedral;
    } else {
      for (j = 0; j < work_atom->valence; j++) {
      if (is_qcode_match(work_atom->bond[j],
               second_atom, tolerance)) {
        work_dihedral.list_index = i;
        work_dihedral.atom1_offset =
        get_atom_offset(work_atom);
        work_dihedral.atom2_offset =
        get_atom_offset(work_atom->bond[j]);
        return work_dihedral;
      }
      }
    }
    }
    work_atom = work_atom->next;
  }
  }

  return work_dihedral;
}

/* The following function takes an atom * and a list of atoms, and adds */
/* the atom * to the end of the list, resetting the last element of the */
/* list to NULL (the sentry value).  It returns a pointer to the list.  */
/* it is very imortant to catch this pointer from the calling function, */
/* since realloc will happily place the array somewhere in memory not */
/* at the same place as it was orginally called */
atom **add_to_molecule_list(atom *some_atom, atom **list) {

  int old_list_size;

  if (list == NULL) {
  /* We need to set up space for it initially */
  if ( (list = malloc ( 1 * sizeof (atom *) ) ) == NULL ) {
    error_exit("Cannot allocate base of list in "
         "add_to_molecule_list()");
  }
  /* And set the end of list sentry */
  list[0] = NULL;
  }

  /* Error checking */
  if (some_atom == NULL ) {
  /* just return the original list */
  return list;
  }

  /* First, find out where the last item in the list is */
  for (old_list_size = 0; list[old_list_size] != NULL; old_list_size++) { }
  old_list_size += 1;

  /* Now, realloc space in the list */
  if ((list = realloc( list, (old_list_size + 1) * sizeof (atom *))) == NULL) {
  warn_out("Unable to expand list in add_to_molecule_list(), this "
       "will likely be fatal");
  return list;
  }

  /* Add the new atom to the list, and re-mark the end of list sentry */
  list[old_list_size - 1] = some_atom;
  list[old_list_size - 0] = NULL;

  return list;
}

/* The following function adds the indicated dihedral to a list.  It takes */
/* the base of the dihedral list (which, if it's NULL, it allocates), the */
/* base of a list of molecules (i.e., frag_list, etc.), and two atoms.  */
/* It will only add the indicated bond if the resulting dihedral is */
/* 'interesting' - the sole criterium for that at present is that it is */
/* not a terminal methyl group; other rules can be added as needed.  It */
/* will also not add a duplicate entry.  Like add_to_list(), it is */
/* important to retrieve the pointer in the calling function, since the */
/* realloc call may move the base of the array */
dihed *add_dihedral(dihed* dihedral_list, atom **molecule_list,
          atom *atom1, atom *atom2) {

  int i, dlist_size, offset1, offset2, molecule_list_offset;
  atom* molecule_base;

  /* Basic error checking */
  if ( molecule_list == NULL ) {
  warn_out("NULL molecule list passed to add_dihedral(), this may be fatal");
  return dihedral_list;
  }
  if (atom1 == NULL || atom2 == NULL) {
  warn_out("NULL pointer instead of atom * type passed to add_dihedral(), "
       "this will likely be fatal");
  return dihedral_list;
  }
  /* Make sure the atoms are from the same fragment */
  offset1 = get_atom_offset(atom1);
  offset2 = get_atom_offset(atom2);
  if ( (atom1 - offset1) != (atom2 - offset2) ) {
  warn_out("atoms from different molecules passed to add_dihedral(), "
       "returning dihedral list unchanged");
  return dihedral_list;
  }

  molecule_list_offset = -1;
  /* Find the molecule_list_offset */
  molecule_base = atom1 - offset1;
  for (i = 0; molecule_list[i] != NULL; i++) {
  if (molecule_base ==
    (molecule_list[i] - get_atom_offset(molecule_list[i]) ) ) {
    molecule_list_offset = i;
    break;
  }
  }

  if (molecule_list_offset == -1 ) {
  warn_out("molecule passed to add_dihedrel (implicily) is not a member "
       "of the molecule list!" );
  return dihedral_list;
  }

  /* If the passed item is a duplicate, return the original list */
  if ( dihedral_list == NULL ) {
  i = 0;
  } else {
  for (i = 0; dihedral_list[i].list_index != -1; i++) {
    if (
      dihedral_list[i].list_index == molecule_list_offset &&
      (
       ( dihedral_list[i].atom1_offset == offset1 &&
       dihedral_list[i].atom2_offset == offset2 )
       ||
       ( dihedral_list[i].atom1_offset == offset2 &&
       dihedral_list[i].atom2_offset == offset1 )
       )
      ) {
    /* Return the original list, doing nothing */
    return dihedral_list;
    }
  }
  }
  dlist_size = i;

  /* If either of the atoms have a valence of one, return the original list */
  if ( atom1->valence <= 1 ) { return dihedral_list; }
  if ( atom2->valence <= 1 ) { return dihedral_list; }

  /* If either is a terminal methyl group, return the original list */
  if ( is_methyl_group(atom1) ) { return dihedral_list; }
  if ( is_methyl_group(atom2) ) { return dihedral_list; }

  /* Finally, we can expand the existing dihedral list */
```

```
  if ( (dihedral_list = realloc( dihedral_list, (dlist_size + 2) *
                sizeof(dihed) ) ) == NULL ) {
  warn_out("could not reallocate space for the dihedral_list in "
       "add_dihedral()" );
  return dihedral_list;
  }

  /* Reset the sentry value */
  dihedral_list[dlist_size + 1].list_index = -1;
  dihedral_list[dlist_size + 1].atom1_offset = -1;
  dihedral_list[dlist_size + 1].atom2_offset = -1;

  /* And ... copy the relevant values */
  dihedral_list[dlist_size].list_index = molecule_list_offset;
  dihedral_list[dlist_size].atom1_offset = offset1;
  dihedral_list[dlist_size].atom2_offset = offset2;

  return dihedral_list;
}

/* This is a complete re-write of the original function, since it needed */
/* to use internet sockets.  The meanings of the passed parameters follow: */
/* *some_atom and *second_atom are the atoms to find qcode matches for */
/* in the database.  If *second_atom == NULL, we seek an atom match, */
/* and if it is not NULL, we are seeking a bond match.  The algorithm */
/* will be somewhat similar to the old implementation, but will use the */
/* qdb_query_server.pl to get the best match.  The function returns a */
/* qdb_result structure (one or more), which contains the directory in */
/* which a match was found, the offsets (within that db directory) that */
/* correspond to the atoms given in some_atom and second_atom, and their */
/* respective tolerances.  On failure, it simply returns an 'empty' */
/* qdb_result, whose members are all -1.  The full algorithm is outlined */
/* as follow: */
/* 1)  Query the query server for the best match to either the atom */
/*     or bond specified */
/* 2)  Format the qdb_result for returning */
/* 3)  Return it! */

qdb_result is_in_qdb(atom *some_atom, atom *second_atom, float tolerance,
           int socket_handle) {

  /* Initializations */
  int i, length;
  char work_qcode[MAXSTR * 8], work_qcode2[MAXSTR * 8];
  char *response_p, send_string[MAXSTR * 16];
  qdb_result return_result;

  /* Initialize empty return_result for error returns */
  BLANK_QDB_RESULT(return_result);

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  /* Error checking */
  if (some_atom == NULL) {
  warn_out("Null pointer passed as first argument to is_in_qdb()");
  return return_result;
  }

  if (some_atom == second_atom) {
  warn_out("Identical atoms passed to is_in_qdb().  This was most likely"
       "unintended, and may be fatal");
  }

  if (tolerance <= 0) {
  warn_out("Invalid tolerance passed to is_in_qdb()");
  return return_result;
  }

  response_p = NULL;

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  if ( second_atom == NULL ) {
  /* An atom match was requested */

  /* Prepare qcode for input */
  work_qcode[0] = '\0';
  sprintf(work_qcode, "%.255Lg", (some_atom->qcode)[0]);
  for (i = 1; i < QDEPTH; i++) {
    sprintf(work_qcode, "%s %.255Lg",
        work_qcode, (some_atom->qcode)[i]);
  }

  sprintf(send_string, "get atom match {%s}", work_qcode);

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  if (length = strlen(send_string),
    !sendall(socket_handle, send_string, &length)){
    error_exit("Unable to finish communication with server in "
         "is_in_qdb()");
  }

  socket_finish_send(socket_handle);

  response_p = recvall(socket_handle);
  if ( !response_p ) {
    error_exit ("Receive buffer overflow error in is_in_qdb(), "
          "increase the buffer size in recvall() to 'fix'"
          "this problem");
  }

  if ( strlen(response_p) ) {
    int match_count = 0;
    match_count =
    sscanf(response_p, "%s%i%f",
         return_result.directory,
         &return_result.atom1,
         &return_result.tolerance1
         );
    if ( match_count != 3 ) {
    error_exit("Failed to format return string from database "
           "query in is_in_qdb()");
    }
  }

  } else {
  /* A bond match was requested */

  /* Prepare qcodes for input */
  work_qcode[0] = '\0';
  sprintf(work_qcode, "%.255Lg", (some_atom->qcode)[0]);
  for (i = 1; i < QDEPTH; i++) {
    sprintf(work_qcode, "%s %.255Lg",
        work_qcode, (some_atom->qcode)[i]);
  }
  work_qcode2[0] = '\0';
  sprintf(work_qcode2, "%.255Lg", (second_atom->qcode)[0]);
  for (i = 1; i < QDEPTH; i++) {
    sprintf(work_qcode2, "%s %.255Lg",
        work_qcode2, (second_atom->qcode)[i]);
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  sprintf(send_string, "get bond match {%s} {%s}", work_qcode, work_qcode2);

  if (length = strlen(send_string),
    !sendall(socket_handle, send_string, &length)){
    error_exit("Unable to finish communication with server in "
         "is_in_qdb()");
  }

  socket_finish_send(socket_handle);

  response_p = recvall(socket_handle);
  if ( !response_p ) {
    error_exit ("Receive buffer overflow error in is_in_qdb(), "
          "increase the buffer size in recvall() to 'fix'"
          "this problem");
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  if ( strlen(response_p) ) {
    int match_count = 0;
    match_count =
    sscanf(response_p, "%s%i%i%f%f",
         return_result.directory,
         &return_result.atom1,
         &return_result.atom2,
         &return_result.tolerance1,
         &return_result.tolerance2
         );
    if ( match_count != 5 ) {
    error_exit("Failed to format return string from database "
           "query in is_in_qdb()");
    }
  }

  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  return return_result;
}

/* The following function is responsible for making any 'special queries' */
/* to the database server.  As it stands, the main task of the server is */
/* to report if a match for an atom or bond is in the quantum chemistry */
/* database.  If it is, it returns with the directory and atoms that */
/* match.  Other special queries may become necessary, and this function */
/* is a 'catchall' for new queries that need to be added.  The meaningns */
/* of the parameters follow: */
/* atom1          An atom of interest, may be null (but this may or may not */
/*                make sense with the supplied op_code */
/* atom2          An atom of interest, may be null (but this may or may not */
/*                make sense with the supplied op_code) */
/* socket_handle  A socket_handle that is open and ready */
/* other          A very generic pointer to any other data the function */
/*                may need */
/* op_code        An integer specifying the mode of operation, legal */
/*                values are: */
/*    0   ->   Ask the server about the environment around the matched */
/*             atom.  Both atom1 and atom2 must be null, and the */
/*             other string will contain all of the information.  Its */
/*             format is as follows: */
/* <directory_name> <atom #> <tolerance> <distance from atom#> <qcode> */
/*     [distance from atom#> <qcode> ] ... */
/* */
/*             In this fasion, we can query about the position of */
/*             numerous asymmetric carbons around the 'epi-center', */
/*             and return a non-null pointer if there was indeed a */
/*             match */

char *qdb_special_query(atom *atom1, atom* atom2, int socket_handle,
            int op_code, void *other) {

  char send_string[MAXSTR * 16], *response_p;
  int length;

  switch (op_code) {

  case 0:
  /* Query about asymmetry environment */
```

```
  sprintf(send_string, "get long query %d", strlen("asymmetry ") +
      strlen((char*)other) + 2 /* For terminator */ );
  if (length = strlen(send_string),
    !sendall(socket_handle, send_string, &length)){
    error_exit("Unable to finish communication with server in "
         "is_in_qdb()");
  }
  socket_finish_send(socket_handle);

  sprintf(send_string, "asymmetry %s", (char *)other);
  if (length = strlen(send_string),
    !sendall(socket_handle, send_string, &length)){
    error_exit("Unable to finish communication with server in "
         "is_in_qdb()");
  }
  socket_finish_send(socket_handle);

  response_p = recvall(socket_handle);
  if ( !response_p ) {
    error_exit ("Receive buffer overflow error in is_in_qdb(), "
          "increase the buffer size in recvall() to 'fix'"
          "this problem");
  }

  return response_p;
  default:
  error_exit("Unknown op_code passed to qdb_special_query()");
  }

  return NULL;
}

/* The following function prepares an assymetry query string.  Its */
/* location as a separate function is primarily to keep main() uncluttered */
/* The requred format is: */
/* <directory_name> <atom #> <tolerance> <distance from atom#> <qcode> */
/*     [distance from atom#> <qcode> ] ... */
/* Other notes: */
/* Many of the sprintf's could have been replace by strcat's or strncat's */
/* This function has only been tested with 1 assymetric atom.  I've tried */
/* to put enough catches into it so if there are logical errors, it will */
/* die with more assymetric carbons, but as I said, this hasn't been */
/* tested, and will not be until the case arises */
char *prepare_asymmetry_query(qdb_result work_result, atom *center,
                float tolerance, int asymmetry_range) {

  char work_string[MAXSTR * 8], work_string_2[MAXSTR];
  atom *work_atom, *molecule_base;
  int i, total_length = 0, alloc_length = MAXSTR;
  static char* return_string = NULL;
  boolean expand = false;

  /* Note:  This function makes a _lot_ of assumptions, and is not well */
  /* used unless you've recently called both recurse_atom_do(), and have */
  /* a sensible result from the database query server for work_result. */
  /* Due to it's purpose (very specific) error checking will be minimal */

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  /* Free memory if requested */
  if (asymmetry_range < 0) {
  if (asymmetry_range == -999) {
    if (return_string) {
    free(return_string);
    }
    return NULL;
  } else {
    /* Negative asymmetry range makes no sense */
    error_exit("Negative asymmetry_range passed to "
         "prepare_asymmetry_query(), this is senseless.  Pass "
         "-999 if you wish to free the function's work string\n");
  }
  }

  /* Error checking (as much as we'll be doing) */
  if (strlen(work_result.directory) == 0) {
  error_exit("Uninitialized (zero_length directory) qdb_result passed "
         "to prepare_asymmetry_query()");
  }
  if (work_result.atom1 == -1) {
  error_exit("Uninitialized (atom1 = -1) qdb_result passed "
         "to prepare_asymmetry_query()");
  }
  if (!center) {
  error_exit("Null atom pointer passed to prepare_asymmetry_query()");
  }

  /* We kept return_string static so we could free it ourselves between */
  /* calls */
  if (return_string) {
  free(return_string);
  }

  /* Initialize return string */
  if ( !(return_string = malloc(alloc_length * sizeof(char) ) ) ) {
  error_exit("Unable to initialize space for return string in "
         "prepare_asymmetry_query()");
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  tolerance = reduce_tolerance(tolerance, asymmetry_range);

  sprintf(work_string, "%g", tolerance);

  total_length += strlen(work_result.directory) + 1  /* space */ +
  sprintf(work_string_2, "%d", work_result.atom1) + 1 +
  strlen(work_string);  /* Note the lack of a + 1 at the end.  That was */
                        /* there to account for the '\0', but this is */
                        /* not reported by sprintf */

  while ( (total_length + 1 ) > alloc_length) {
  alloc_length += MAXSTR;
  expand = true;
  }

  if (expand) {
  if ( !(return_string = realloc(return_string,
                 alloc_length * sizeof(char) ) ) ) {
    error_exit("Unable to resize (#1) return string in "
         "prepare_asymmetry_query()");
  }
  expand = false;
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  return_string[0] = '\0';

  if ( sprintf(return_string, "%s %d %g", work_result.directory,
         work_result.atom1, tolerance) != total_length ) {
  error_exit("sprintf() count failed to match total_length (#1) "
         "in prepare_asymmetry_query()");
  }

  molecule_base = molecule_return_base(center);

  while ( ( work_atom = atom_list_manage(center, A_COUNT) ) != NULL &&
      get_atom_offset(work_atom) ) {
  work_atom = atom_list_manage(NULL, A_POP);

  /* Prepare qcode */
  sprintf(work_string, "{%.255Lg", (work_atom->qcode)[0]);
  for (i = 1; i < QDEPTH; i++) {
    sprintf(work_string, "%s %.255Lg", work_string,
        (work_atom->qcode)[i]);
  }
  sprintf(work_string, "%s} ", work_string);

  total_length = strlen(return_string) + 1 +
    sprintf(work_string_2, "%ld", work_atom->state) + 1 +
    strlen(work_string);

  while ( (total_length + 1) > alloc_length) {
    alloc_length += MAXSTR;
    expand = true;
  }
  if (expand) {
    if ( !(return_string = realloc(return_string,
                 alloc_length * sizeof(char) ) ) ) {
    error_exit("Unable to resize return string (#2) in "
           "prepare_asymmetry_query()");
    }
    expand = false;
  }

  if ( sprintf(return_string, "%s %ld %s", return_string,
         work_atom->state, work_string)
     != total_length ) {
    error_exit("sprintf() count failed to match total_length (#2) "
         "in prepare_asymmetry_query()");
  }
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  return return_string;
}

/* The following function reduces the tolerance for searching for matches */
/* to asymmetric carbons some atoms removed from the central atom. */
float reduce_tolerance(float tolerance, int range) {

  /* After working (practically) with the qcode matching of remote
     qcodes, it has become clear that we _cannot_ expect the remote
     asymmetric carbon to match as exactly as we matched the atom when
     we put the fragment into the database.  The following rules I'm
     writing are completely arbitrary, and could potentially be put
     into .qdb_checkrc as configuration of some sort or another.  For
     now, we'll just live with my arbitraryness */

  /* If we can reduce the exact match by it's distance from the central atom */
  /* do so, otherwise, make the exact match 0 */

  if (tolerance >= range) {
  tolerance -= range;
  } else {
  tolerance -= (int)tolerance;
  }

  /* Reduce the fractional match by 20% of it's original value.  This is */
  /* probably too aggressive (but maybe not?), and we can tune the number */
  /* at a later point */
  tolerance -= ( (tolerance - (int)tolerance) * 0.20 ) ;

  return tolerance;

}

/* The following function adds a result to the list.  It requires a */
/* qdb_result that's already formatted, since it simply copies the values */
/* from the original result.  The flag is copied to the tolerance1 member */
/* of the new result (at the end of the list we're returning), and */
/* indicates where the result came from.  The meanings of the flags are as */
/* follows:                                                                 */
/* 1 ---> frag_list                                                         */
/* 2 ---> qdb_result_list                                                   */
/* That's it!  Those are the only two places we'll be keeping results.  */
qdb_result **add_dbresult_to_qdb_result_list(qdb_result **original_list,
                       int flag, qdb_result my_result) {
```

```
  int list_tail;
  qdb_result *work_result;

  /* Error checking */

  if (original_list == NULL) {
  /* We need to set up space for it initially */
  if ( (original_list = malloc ( 1 * sizeof (qdb_result *) ) ) == NULL ) {
    error_exit("Cannot allocate base of list in "
        "add_dbresult_to_qdb_result_list()");
  }
  /* And set the end of list sentry */
  original_list[0] = NULL;
  }

  /* First, find out where the last item in the list is */
  for (list_tail = 0; original_list[list_tail] != NULL;
     list_tail++) { }
  list_tail += 1;

  /* Now, realloc space in the list */
  if ((original_list = realloc( original_list,
              (list_tail + 1) * sizeof (qdb_result *))) == NULL) {
  warn_out("Unable to expand list in add_dbresult_to_qdb_result_list(), "
      "this will likely be fatal");
  return original_list;
  }

  /* Allocate space for the new qdb_result */
  if ( (work_result = malloc ( sizeof(qdb_result) ) ) == NULL ) {
  error_exit("Unable to malloc space for new qdb_result in "
        "add_dbresult_to_qdb_result_list()");
  }

  /* Add the new atom to the list, and re-mark the end of list sentry */
  BLANK_QDB_RESULT((*work_result));
  strcpy(work_result->directory, my_result.directory);
  work_result->atom1 = my_result.atom1;
  work_result->atom2 = my_result.atom2;
  work_result->tolerance1 = (float)flag;
  work_result->tolerance2 = my_result.tolerance2;
  work_result->s_atom1 = my_result.s_atom1;
  work_result->s_atom2 = my_result.s_atom2;
  work_result->offset = my_result.offset;

  original_list[list_tail - 1] = work_result;
  original_list[list_tail - 0] = NULL;

  return original_list;
}

/* Function mania!  Beginning here, I'll be making efforts to migrate */
/* 'messy' looking function bits out of main(), and into here.  There */
/* are a lot of bits of code that really only rely on a few variables, */
/* and they really belong here, to keep main() readable */

/* The following function returns the requested value from the rc_file */
/* specified.  Note that the rc_file must already be opened by the */
/* calling environment.  The function either returns the value requested, */
/* or, if it can't find it, the function returns the null string ("").  */

char *get_rcfile_resource(FILE *rc_file, const char* id) {

  static char return_string[MAXSTR];
  char work_string[MAXSTR];

  strcpy(return_string, "");
  strcpy(work_string, "");

  rewind(rc_file);

  while ( fgets(work_string, MAXSTR, rc_file) != NULL ) {
  if ( strncmp( work_string, id, strlen(id) ) == 0 ) {
    if ( (fscanf(rc_file, "%s", work_string) ) != 1 ) {
    warn_out ("Unable to get resource from rc_file, this "
         "may be fatal");
    } else {
    break;
    }
  }
  }

  strcpy(return_string, work_string);

  return return_string;
}

/* The function: atom *read_init_formatted_coordinates(FILE
 *read_stream) has been moved to atom_handling.c.  All programs that
 use this function have not been thoroughly tested */

/* The function: void read_formatted_connectivity(FILE *read_stream,
   atom *member)has been moved to atom_handling.c.  All programs that
   use this function have not been thoroughly tested */

/* The following function is one of the 'moved out of main' class, but */
/* in this case out of pure necessity, due mainly to the convuluted */
/* logic involved, though I'll do my best to document here as thoroughtly */
/* as possible.  What it does is take a (already formatted) qdb_result, */
/* which has as members descriptions of what we're looking for.  It */
/* takes a pointer to somewhere in the parent molecule, and a pointer */
/* to the list containing the fragment to compare to the parent molecule.  */
/* Finally, we need the range and tolerance to make the real comparisons. */
/* Unfortunately, we can't really take advantage of the flexibility of */
/* recurse_molecule_do(), since we need to see the states from this */
/* function, but we need to restore them before returning to the calling */
/* environment.  We are not concerned for the fragment's states, however */
int compare_asymmetry_environment_atom(qdb_result result,
                    atom *parent_molecule, atom **compare_list, int range,
float tolerance) {

  /* Variable declarations */
  atom *p_atom1, *p_atom2, *f_atom1, *f_atom2, *fragment_base;
atom **p_atom_asymmetric_neighbors, **f_atom_asymmetric_neighbors;
atom *work_atom;
int p_list_size, f_list_size;
int p_atom1_neighbor_count, p_atom2_neighbor_count, f_atom1_neighbor_count,
f_atom2_neighbor_count;
long int *p_states;
float reduced_tolerance;
int i;
boolean is_bond, are_homomers, are_enantiomers;
int (*fcnptr) (atom *);

/* Error checking */
if (result.atom1 == -1) {
error_exit("result.atom1 == -1 in compare_asymmetry_environment_atom(), "
       "this indicates an uninitialized result was passed to this "
       "function, and is a fatal error\n");
}

if (result.s_atom1 == -1) {
error_exit("result.s_atom1 == -1 in compare_asymmetry_environment_atom(), "
       "this indicates an uninitialized result was passed to this "
       "function, and is a fatal error\n");
}

if ( parent_molecule == NULL ) {
error_exit("NULL pointer received for parent molecule in "
       "compare_asymmetry_environment_atom()");
}

if ( compare_list == NULL ) {
error_exit("NULL pointer received for compare list in "
       "compare_asymmetry_environment_atom()");
}
if ( result.offset < 0 ) {
error_exit("Non-integral or negative value stored in offset in "
       "compare_asymmetry_environment_atom()");
}

/* We probably _should_ do bounds checking for result.atom1, result.atom2, */
/* result.s_atom1, result.s_atom2, and result.tolerance1, and offset ,but */
/* this is currently given a low priority */


/* Variable initializations */

/* Determine whether we're talking about a bond or an atom to look around */
if ( result.atom2 == -1 && result.s_atom2 == -1 ) {
is_bond = false;
} else if ( result.atom2 == -1 || result.s_atom2 == -1 ) {
/* This is like an xor (with the previous if), and indicates an error */
/* condition */
error_exit("In compare_asymmetry_environment_atom(), the second atoms "
       "in the result were 1/2 marked as occupied, and 1/2 emptry. "
       "This indicates that either atom2 or s_atom2 is trying "
       "to indicate we have a bond, and the other is indicating "
       "we have an atom.  This is a logically impossible condition ");
} else {
is_bond = true;
}

/* Initialize parent atoms */
i = get_atom_offset(parent_molecule);
p_atom1 = parent_molecule - i + result.s_atom1;
p_atom2 = is_bond ? (parent_molecule - i + result.s_atom2) : NULL;

/* Initialize fragment atoms */
fragment_base = compare_list[(int)result.offset] -
get_atom_offset(compare_list[(int)result.offset]);
f_atom1 = fragment_base + result.atom1;
f_atom2 = is_bond ? (fragment_base + result.atom2) : NULL;

/* Initialize function pointer for usage in recurse_molecule_do() */
fcnptr = i_does_it_have_s_descriptor;

/* Save our states */
p_states = save_states(p_atom1);

p_atom_asymmetric_neighbors = f_atom_asymmetric_neighbors = NULL;
p_list_size = f_list_size = 0;

p_atom1_neighbor_count = p_atom2_neighbor_count = 0;
f_atom1_neighbor_count = f_atom2_neighbor_count = 0;

/* All of the variables are initialized, and we're ready to start
   poking around in an attempt to determine exactly what we should
   return.  The following design represents a serious change in
   philosophy.  Instead of going into deeply nested logic, we'll do
   the recurse_molecule_do to each of the four atoms (only two if
   we're not looking at a bond), and cache all of the results (the
   atoms' identities and states) in the lists initialized above.
   After we've got all of this information, only then do we actually
   do the comparisons to see if we have identical or enantiomeric
   environments */

if ( recurse_molecule_do (fcnptr, p_atom1, range, 0) ) {
/* We need to create the lists */
if ( ( work_atom = atom_list_manage(p_atom1, A_COUNT) ) == NULL ) {
  /* NULL from that function indicates that the molecule that holds */
  /* the atoms in the list isn't big enough to hold the count, */
  /* this is definately an error, but how it might happen is */
  /* a complete mystery :-) */
  error_exit("List out of bounds in "
       "compare_asymmetry_environment_atom()");
}

p_atom1_neighbor_count = p_list_size =
  get_atom_offset( atom_list_manage(p_atom1, A_COUNT) );

/* We have the size, now let's allocate the space and populate the */
/* two relevant lists */
if ( p_list_size && ( ( p_atom_asymmetric_neighbors =
           malloc ( p_list_size * sizeof (atom *) ) ) == NULL ) ) {
  error_exit("Unable to allocate memory for "
       "p_atom_asymmetric_neighbors in "
```

```
       "compare_asymmetry_environment_atom()");
}

for ( i = 0; i < p_list_size; i++ ) {
  work_atom = atom_list_manage(NULL, A_POP);
  if ( work_atom == NULL ) {
  error_exit("Atom list ran out of members to pop before "
         "anticipated in "
         "compare_asymmetry_environment_atom()");
  }
  p_atom_asymmetric_neighbors[i] = work_atom;
}
}

/* If we have a bond, let's expand the information we have on the parent */
/* atoms */
if ( is_bond && recurse_molecule_do (fcnptr, p_atom2, range, 0) ) {
/* We need to expand the lists */
if ( ( work_atom = atom_list_manage(p_atom1, A_COUNT) ) == NULL ) {
  /* NULL from that function indicates that the molecule that */
  /* holds the atoms in the list isn't big enough to hold the count, */
  /* this is definately an error, but how it might happen is */
  /* a complete mystery :-) */
  error_exit("List out of bounds in "
       "compare_asymmetry_environment_atom()");
}

p_atom2_neighbor_count = get_atom_offset(
                     atom_list_manage(p_atom2, A_COUNT) );
p_list_size += p_atom2_neighbor_count;

/* We have the size, now let's allocate the space and populate the */
/* two relevant lists */
if ( (p_list_size ) &&
   ( ( p_atom_asymmetric_neighbors =
     realloc ( p_atom_asymmetric_neighbors,
          p_list_size * sizeof (atom *) ) ) == NULL ) ) {
  error_exit("Unable to expand memory for "
       "p_atom_asymmetric_neighbors in "
       "compare_asymmetry_environment_atom()");
}

for ( i = p_atom1_neighbor_count; i < p_list_size; i++ ) {
  work_atom = atom_list_manage(NULL, A_POP);
  if ( work_atom == NULL ) {
  error_exit("Atom list ran out of members to pop before "
         "anticipated in "
         "compare_asymmetry_environment_atom()");
  }
  p_atom_asymmetric_neighbors[i] = work_atom;
}
}

/* End parent atom list initializations */

/* Begin fragment atom list initializations */

if ( recurse_molecule_do (fcnptr, f_atom1, range, 0) ) {
/* We need to create the lists */
if ( ( work_atom = atom_list_manage(p_atom1, A_COUNT) ) == NULL) {
  /* NULL from that function indicates that the molecule that holds */
  /* the atoms in the list isn't big enough to hold the count, */
  /* this is definately an error, but how it might happen is */
  /* a complete mystery :-) */
  error_exit("List out of bounds in "
       "compare_asymmetry_environment_atom()");
}

f_atom1_neighbor_count = f_list_size =
  get_atom_offset( atom_list_manage(p_atom1, A_COUNT) );

/* We have the size, now let's allocate the space and populate the */
/* two relevant lists */
if ( f_list_size && ( ( f_atom_asymmetric_neighbors =
           malloc ( f_list_size * sizeof (atom *) ) ) == NULL ) ) {
  error_exit("Unable to allocate memory for "
       "f_atom_asymmetric_neighbors in "
       "compare_asymmetry_environment_atom()");
}

for ( i = 0; i < f_list_size; i++ ) {
  work_atom = atom_list_manage(NULL, A_POP);
  if ( work_atom == NULL ) {
  error_exit("Atom list ran out of members to pop before "
         "anticipated in "
         "compare_asymmetry_environment_atom()");
  }
  f_atom_asymmetric_neighbors[i] = work_atom;
}
}

/* If we have a bond, let's expand the information we have on the parent */
/* atoms */
if ( is_bond && recurse_molecule_do (fcnptr, f_atom2, range, 0) ) {
/* We need to expand the lists */
if ( atom_list_manage(p_atom1, A_COUNT) == NULL ) {
  /* NULL from that function indicates that the molecule that holds */
  /* the atoms in the list isn't big enough to hold the count, */
  /* this is definately an error, but how it might happen is */
  /* a complete mystery :-) */
  error_exit("List out of bounds in "
       "compare_asymmetry_environment_atom()");
}

f_atom2_neighbor_count = get_atom_offset(
                     atom_list_manage(p_atom1, A_COUNT) );
f_list_size += f_atom2_neighbor_count;

/* We have the size, now let's allocate the space and populate the */
/* two relevant lists */
if ( (f_list_size ) &&
   ( ( f_atom_asymmetric_neighbors =
     realloc ( f_atom_asymmetric_neighbors,
          f_list_size * sizeof (atom *) ) ) == NULL ) ) {
```

```
  error_exit("Unable to expand memory for "
       "f_atom_asymmetric_neighbors in "
       "compare_asymmetry_environment_atom()");
}

for ( i = f_atom1_neighbor_count; i < f_list_size; i++ ) {
  work_atom = atom_list_manage(NULL, A_POP);
  if ( work_atom == NULL ) {
  error_exit("Atom list ran out of members to pop before "
         "anticipated in "
         "compare_asymmetry_environment_atom()");
  }
  f_atom_asymmetric_neighbors[i] = work_atom;
}
}

/* Ok, we have tons of lists and numbers, and we can begin all of our */
/* tests to try to determine what we should be returning.  Remember, */
/* before returning anything, we need to free the new memory we */
/* allocated.  */

/* If both lists are empty, there's no asymmetry to speak of, and we */
/* can return */
if ( p_list_size == 0 && f_list_size == 0 ) {
/* In this case, there was no memory allocated, but we should still */
/* restore the states (which also frees the memory originally */
/* allocated for them */
restore_states(p_atom1, p_states);
return 1;
}

/* If the lists are of unequal sizes, there is a different environment */
/* around each one, and we don't have a match */
if ( p_list_size != f_list_size ) {
if ( p_atom_asymmetric_neighbors ) { free(p_atom_asymmetric_neighbors); }
if ( f_atom_asymmetric_neighbors ) { free(f_atom_asymmetric_neighbors); }
restore_states(p_atom1, p_states);
return 0;
}

/* If we haven't returned yet, it's because both lists have some members, */
/* and further, they have the same number of members.  The last check of */
/* this type is to make certain that we have the same number of matches */
/* on each of the atoms indicated */
if (p_atom1_neighbor_count != f_atom1_neighbor_count) {
if ( p_atom_asymmetric_neighbors ) { free(p_atom_asymmetric_neighbors); }
if ( f_atom_asymmetric_neighbors ) { free(f_atom_asymmetric_neighbors); }
restore_states(p_atom1, p_states);
return 0;
}

/* Finally, we're ready to compare and try to determine if the lists */
/* represent homomers, enantiomers, or diastereomers */

/* Note:  The following matching algorithm is not 'smart', i.e., it's */
/* possible, but highly unlikely, that it will return a false negative.  */
/* For example, if atom a has an R and an S atom at the same distance */
/* from it, with the same (or _very_ similar qcodes, both the */
/* are_entantiomers, and are_homomers flags will be set.  On exit from */
/* the search, the program will conclude that the two molecules have a */
/* diastereomeric relationship, and will return a false negative.  A */
/* 'smarter' search algorithm could be implemented, but since this is an */
/* extremely unlikely case, it will not be, until the issue arises.  Note */
/* that in the worse case (with the current program design), the program */
/* would submit duplicate fragments to have calculations run on, but in */
/* subsequent runs, the information would be pulled from the database, */
/* (at least that's what I _think_ would happen */
are_homomers = are_enantiomers = false;
for (i = 0; i < p_list_size; i++) {
/* This sends us through the loop, remember p_list_size is equal to */
/* f_list_size.  Also, we'll declare our own local variables for */
/* working within this loop. */
int j, loop_init, loop_max;

/* We don't know that the orders of the member of the lists are */
/* the same, so we need to search each section for the corresponding */
/* member in the second list */
if ( i < f_atom1_neighbor_count ) {
  /* Loop from beginning */
  loop_init = 0;
  loop_max = f_atom1_neighbor_count;
} else {
  /* Loop from f_atom1_neighbor_count */
  loop_init = f_atom1_neighbor_count;
  loop_max = f_list_size;
}

for (j = loop_init; j < loop_max; j++) {
  /* We need the states to match (distance from 'epicenter') */
  /* and for it to be a qcode match with the reduced tolerance */
  reduced_tolerance = reduce_tolerance(tolerance,
                  (f_atom_asymmetric_neighbors[j])->state);
  if ( (f_atom_asymmetric_neighbors[j])->state ==
     (p_atom_asymmetric_neighbors[i])->state &&
     is_qcode_match( f_atom_asymmetric_neighbors[j],
           p_atom_asymmetric_neighbors[i], reduced_tolerance
           )
     ) {
  /* We found an atom that could be a match, check the */
  /* descriptors on the two atoms and mark the appropriate */
  /* flags.  Note that any atoms we see here should have */
  /* _some_ stereochemical descriptor.  If either does not, */
  /* exit catastrophically */
  if ((f_atom_asymmetric_neighbors[j])->s_descriptor == '\0'
    ||
    (p_atom_asymmetric_neighbors[i])->s_descriptor == '\0') {
    error_exit("Some stereochemical descriptor expected in "
        "compare_asymmetry_environment_atom(), "
        "but none found");
  }

  /* If they're equal, mark the homomers flag */
  if ((f_atom_asymmetric_neighbors[j])->s_descriptor ==
    (p_atom_asymmetric_neighbors[i])->s_descriptor) {
```

```
      are_homomers = true;
    } else {
      /* They must be enantiomers */
      are_enantiomers = true;
    }
    }
  }
  }

  /* Phew!  Our work is done, return whatever is appropriate, but first, */
  /* free any memory we need to free */
  free(p_atom_asymmetric_neighbors);
  free(f_atom_asymmetric_neighbors);
  restore_states(p_atom1, p_states);

  /* If we found homomeric matches and enantiomeric matches, the atoms should
*/
  /* have neither relationship, but see the (long) note at the beginning of */
  /* the previous section */
  if (are_homomers && are_enantiomers) { return 0; }
  if (are_homomers) { return 1; }
  if (are_enantiomers) { return 2; }
  if (!are_homomers && !are_enantiomers) { return 1; }

  /* And the default case, which should never be reached */
  error_exit("Reached end of compare_asymmetry_environment_atom(), "
       "the logic of the function prohibits this");
  return 0;
}

/* The following function serves the same purpose as the previous, but is */
/* designed to handle potential matches to the database.  As such, it */
/* looks extremely different, but acts basically the same.  Remember, */
/* we mess with the states, so we'll save them and restore them before */
/* returning:  Note:  This is now done by calling recurse_molecule_do() */
/* with the RMD_SAVESTATES flag.  Unfortunately, as I understand it, */
/* this is not a solution, since we need the status of the states to */
/* persist until we call prepare_asymmetry_query.  We'll cache the states */
/* ourselves */
int compare_asymmetry_environment_db(qdb_result result,
                  atom *parent_molecule, int socket_handle, int range, float
tolerance) {

  /* Variable declarations */
  atom *p_atom1, *p_atom2;
  boolean is_bond, are_homomers, are_enantiomers;
  int i;
  long int *p_states;
  int (*fcnptr) (atom *);

  /* Error condition testing */
  if ( parent_molecule == NULL ) {
    error_exit("NULL pointer received for parent molecule in "
           "compare_asymmetry_environment_db()");
  }

  if ( result.s_atom1 == -1 ) {
  error_exit("result.s_atom1 == -1 in compare_asymmetry_environment_db(), "
           "this indicates an uninitialized result was passed to this "
           "function, and is a fatal error\n");
  }

  /* We probably _should_ do bounds checking for result.atom1, result.atom2, */
  /* result.s_atom1 and result.s_atom2, but this is */
  /* currently given a low priority */

  /* Variable initializations */

  if ( result.atom2 == -1 && result.s_atom2 == -1 ) {
  is_bond = false;
  } else if ( result.atom2 == -1 || result.s_atom2 == -1 ) {
  /* This is like an xor (with the previous if), and indicates an error */
  /* condition */
  error_exit("In compare_asymmetry_environment_db(), the second atoms "
       "in the result were 1/2 marked as occupied, and 1/2 emptry. "
       "This indicates that either atom2 or s_atom2 is trying "
       "to indicate we have a bond, and the other is indicating "
       "we have an atom.  This is a logically impossible condition ");
  } else {
  is_bond = true;
  }

  /* Initialize parent atoms */
  i = get_atom_offset(parent_molecule);
  p_atom1 = parent_molecule - i + result.s_atom1;
  p_atom2 = is_bond ? (parent_molecule - i + result.s_atom2) : NULL;

  /* Initialize function pointer for usage in recurse_molecule_do() */
  fcnptr = i_does_it_have_s_descriptor;

  /* Save the states to be restored on exit */
  p_states = save_states(p_atom1);

  /* We don't need the lists like the last function, because we'll simply */
  /* pop the values off the stack as we need them */

  /* Again, like compare_asymmetry_environment_atom(), we'll do a flat */
  /* analysis, note that if recurse_molecule_do returns nothing, we can */
  /* simply return a true value, since there's no asymmetry to worry about */

  are_enantiomers = are_homomers = false;

  if ( recurse_molecule_do (fcnptr, p_atom1, range, 0 ) ) {
  char *char_p;
  float reduced_tolerance;

  /* See if the db atom is similar enough to this one */
  reduced_tolerance = reduce_tolerance(tolerance, range);
  char_p = prepare_asymmetry_query(result, p_atom1,
                  reduced_tolerance, range);

  char_p = qdb_special_query(NULL, NULL, socket_handle, 0, char_p);

  /* If all of the answers were yes ... we found all the matches we */
/* were looking for, we then need to go through and see if the */
/* values of the matches were the same or different than we */
/* requested.  The format of the queries' return is: */
/* "y<space>[r|s] or */
/* n<space>" ... */
/* What the server guarantees is that any character will be */
/* followed by a space, any response is a single character */
/* long, and any y will be followed by a space and a */
/* single character stereochemical descriptor (though the */
/* system is flexible to change in the future, i.e., R-S */
/* Cahn Ingold Prelog stereochemical descriptors */

/* Again, like the previous function, we'll simply go through and */
/* compare the results to what we already got.  Unfortunately, when */
/* we called prepare_asymmetry_query(), we clobbered the list of */
/* atoms in range, so we'll re_create it here */
recurse_molecule_do (fcnptr, p_atom1, range, RMD_SAVESTATES);

/* Now, begin the comparisons */
i = 0;
while (char_p[i] != '\0') {
  atom *work_atom;
  switch (char_p[i]) {
  case 'n':
  /* We need to indicate that we have _not_ found an */
  /* asymmetry match in any range we wanted.  See comments at */
  /* then end of the function for why this works */
  are_enantiomers = are_homomers = true;
  i += 2;
  break;
  case 'y':
  /* Look for the type of match */
  work_atom = atom_list_manage(NULL, A_POP);
  if ( work_atom == NULL ) {
    /* This is bad and should never happen */
    error_exit("atom_list_manage (pop) returned NULL pointer "
         "in compare_asymmetry_environment_db()");
  }
  if ( work_atom->s_descriptor == char_p [i + 2] ) {
    are_homomers = true;
  } else {
    are_enantiomers = true;
  }

  /* And increment i */
  i += 4;
  break;
  default:
  error_exit("Reached 'unreachable' point in switch"
         "statement in compare_asymmetry_environment_db()");
  }
}
}

/* Now, do the same with the second atom */
if ( recurse_molecule_do (fcnptr, p_atom2, range, 0 ) ) {
char *char_p;
float reduced_tolerance;

/* See if the db atom is similar enough to this one */
reduced_tolerance = reduce_tolerance(tolerance, range);

/* There is a bit of a catch here.  We need to set result.atom1 to */
/* result.atom2.  The reason is that prepare_asymmetry_query() is */
/* expecting to build the query based on result.atom1.  Just in case */
/* (in the future) result is used further, we'll restore it after the */
/* query string is built */
i = result.atom1;
result.atom1 = result.atom2;

char_p = prepare_asymmetry_query(result, p_atom2, reduced_tolerance,
                range);

result.atom1 = i;

char_p = qdb_special_query(NULL, NULL, socket_handle, 0, char_p);

/* Again, like the previous function, we'll simply go through and */
/* compare the results to what we already got.  Unfortunately, when */
/* we called prepare_asymmetry_query(), we clobbered the list of */
/* atoms in range, so we'll re_create it here */
recurse_molecule_do (fcnptr, p_atom2, range, RMD_SAVESTATES);

/* Now, begin the comparisons */
i = 0;
while (char_p[i] != '\0') {
  atom *work_atom;
  switch (char_p[i]) {
  case 'n':
  /* We need to indicate that we have _not_ found an */
  /* asymmetry match in any range we wanted.  See comments at */
  /* then end of the function for why this works */
  are_enantiomers = are_homomers = true;
  i += 2;
  break;
  case 'y':
  /* Look for the type of match */
  work_atom = atom_list_manage(NULL, A_POP);
  if ( work_atom == NULL ) {
    /* This is bad and should never happen */
    error_exit("atom_list_manage (pop) returned NULL pointer "
         "in compare_asymmetry_environment_db()");
  }
  if ( work_atom->s_descriptor == char_p [i + 2] ) {
    are_homomers = true;
  } else {
    are_enantiomers = true;
  }

  /* And increment i */
  i += 4;
  break;
  default:
  error_exit("Reached 'unreachable' point in switch"
```

```
          "statement in main()");
    }
  }
  }

  /* All our values are set, first, let's do our exit business, then we'll */
  /* figure out what to return */
  restore_states(p_atom1, p_states);

  /* If we found homomeric matches and enantiomeric matches, the atoms should
*/
  /* have neither relationship, but see the (long) note at the beginning of */
  /* the previous section */
  if (are_homomers && are_enantiomers) { return 0; }
  if (are_homomers) { return 1; }
  if (are_enantiomers) { return 2; }
  if (!are_homomers && !are_enantiomers) { return 1; }

  /* And finally, the default case, which should never be reached */
  error_exit("Reached end of compare_asymmetry_environment_db().  The "
       "logic of this function prohibits this");
  return 0;
}

/* This simply allows the developer to see the members of the */
/* result list passed to it.  It uses labels consistent with the purpose */
/* of the list member, not necessarily the name */
void print_result_list(FILE *dest, qdb_result **list) {

  int i;
  qdb_result *work_result;
  boolean is_atom_match = false;
  char source[MAXSTR];

  /* This function prints the 1 base notation for the members of the list. */
  /* Mainly, this allows direct comparison with an open gaussview window */
  if (list == NULL) {
  fprintf(dest, "This list is empty\n");
  return;
  }

  for (i = 0; list[i] != NULL; i++) {
  work_result = list[i];
  if ( work_result->tolerance1 == 1 ) {
    strcpy(source, "Frag");
  } else if ( work_result->tolerance1 == 2 ) {
    strcpy(source, "Qdb");
  } else {
    error_exit("Result at had an invalid (neither 1 nor 2) value "
         "for tolerance1");
    error_exit("");
  }
  if (strcmp(work_result->directory, "") ) {
    fprintf(dest, "Dir: %s ", work_result->directory);
  }
  if (work_result->atom2 == -1 && work_result->s_atom2 == -1) {
    is_atom_match = true;
  }
  if ( is_atom_match && !(work_result->atom2 == -1 &&
             work_result->s_atom2 == -1) ) {
    error_exit("atom1 and s_atom2 do not both agree that we're on "
         "a bond in print_result_list(), this indicates a mistake, "
         "and should be corrected");
  }
  if ( is_atom_match ) {
    fprintf(dest, "Parent atom %d: ", work_result->s_atom1);
  } else {
    fprintf(dest, "Parent bond %d-%d: ", work_result->s_atom1,
        work_result->s_atom2);
  }
  if ( is_atom_match ) {
    fprintf(dest, "%s atom %d: ", source, work_result->atom1);
  } else {
    fprintf(dest, "%s bond %d-%d: ", source, work_result->atom1,
        work_result->atom2);
  }
  if ( work_result->tolerance1 == 1 ) {
    if (work_result->offset == -1) {
    error_exit("offset == -1 while tolerance1 == 1 in "
           "print_result_list(), this is an error condition");
    }
    fprintf(dest, "frag_list offset %d ", work_result->offset);
  } else if ( work_result->tolerance1 == 2 ) {
    fprintf(dest, "qdb ");
  } else {
    error_exit("tolerance1 != 1 or 2 in print_result_list()");
  }
  if ( work_result->tolerance2 == -1 ) {
    /* Do nothing */
  } else if ( work_result->tolerance2 == 0 ) {
    error_exit("tolerance2 == 0 in print_result_list(), this is "
         "an error condition");
  } else if ( work_result->tolerance2 == 1 ) {
    fprintf(dest, "homo ");
  } else if ( work_result->tolerance2 == 2 ) {
    fprintf(dest, "enantio ");
  } else {
    error_exit("Unknown value for tolerance2 in print_result_list()");
  }
  fprintf(dest, "\n");
  }

  return;
}

/* This is a simple debugging fuction that prints a qdb_result */
void print_qdb_result(qdb_result result) {
  printf("Directory: %s\n", result.directory);
  printf("Source atoms: %d, %d\n", result.s_atom1, result.s_atom2);
  printf("Database atoms: %d, %d\n", result.atom1, result.atom2);
  printf("Tolerances: %g %g\n", result.tolerance1, result.tolerance2);
  printf("Offset: %d\n", result.offset);

}
```

qdb_check.c


```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */
```


```
#include "qdb_check.h"

/* This program takes as arguments a tolerance (float).  The other    */
/* information it gets is a structure from stdin, and it reads the    */
/* database directory from .qdb_checkrc.  It outputs (to stdout)      */
/* directions for ab initio jobs that need to be performed    */

int main(int argc,char *argv[])
{

  /* Variable initialization */

  /* Note to self about the (growing list of) 'global' variables:  This */
  /* list is beginning to look a fair bit too big.  In the future it would */
  /* be an excellent idea to try to localize as many of the following */
  /* variables as possible to the blocks in which they are used.  This is */
  /* a low proirity, but will add immensly to the maintainability of the */
  /* program */
  int i, j, k;
  int socket_handle, query_server_port, asymmetry_range;
  float tolerance;
  char work_string[MAXSTR], query_server_host[MAXSTR];
  atom *work_atom, *molecule_base, **frag_list;
  FILE *tmp_file, *out_file, *message_file;
  dihed *dihedral_list;
  qdb_result work_result, **atom_map_list, **bond_map_list;

  /* Initialization of command line variables */
  if ( argc != 1 ) { Usage(); }

  /* Back when tolerance was a command line option ... */
  /* if ( (tolerance = atof(argv[1])) == 0 ) { Usage(); } */

  /* Get the query_server information from the .qdb_checkrc file */
  if ( (tmp_file = fopen(".qdb_checkrc", "r") ) == NULL ) {
  error_exit("Cannot open the file .qdb_checkrc for reading");
  }

  /* Initialization of resources from .qdb_checkrc */
  strcpy(query_server_host,
     get_rcfile_resource(tmp_file, "#query_server_host"));

  strcpy(work_string,
     get_rcfile_resource(tmp_file, "#query_server_port"));
  if (sscanf(work_string, "%d", &query_server_port) != 1 ) {
  error_exit("Unable to format query_server_port (after read from "
       "rcfile) in main()");
  }

  strcpy(work_string,
     get_rcfile_resource(tmp_file, "#tolerance"));
  if (sscanf(work_string, "%f", &tolerance) != 1 ) {
  error_exit("Unable to format tolerance (after read from "
       "rcfile) in main()");
  }

  strcpy(work_string,
     get_rcfile_resource(tmp_file, "#asymmetry_range"));
  if (sscanf(work_string, "%i", &asymmetry_range) != 1 ) {
  error_exit("Unable to format asymmetry_range (after read from "
       "rcfile) in main()");
  }

  fclose(tmp_file);

  message_file = stderr;
  /* Initialization are finished */

  /* Initialize an array of atoms to represent the molecule */
  molecule_base = read_init_formatted_coordinates(stdin);

  /* Read connectivity if it's provided */
  read_formatted_connectivity(stdin, molecule_base);

  /* Try to verify all of the bonds.  This particular function has been */
  /* worked out reasonably well, and I'd call it alpha quality.  It needs */
  /* to run the gamut, and may need to have special cases handled for */
  /* non-CNHO type molecules.  Regardless, if someone else can provide */
  /* the connectivity, (it skips trying to find bonds if the valence is */
  /* already full, presumably from the previous call to */
  /* read_formatted_connectivity() it works great (definately post beta) */
  if ( !verify_molecule_connectivity(molecule_base) ) {
  /* The connectivity was hosed somehow, so we simply exit */
  error_exit("Verification of input molecule's connectivity failed");
```

```
  }

  /* It has been verified that the connectivity is correct for the molecule */
  /* structure used for development. Also, routines for traversing the */
  /* molecule via an array, a linked list, and recursively are supplied */
  /* in comments after main() */

  /* get qcodes for the molecule */
  if ( assign_qcodes(molecule_base) == 0 ) {
  error_exit("Failed to initialize all qcodes in main()");
  }

  /* Now that we have the qcodes, we can go on to assign any stereochemical */
  /* descriptors that we need */

  /* The following is a (slightly) different way to iterate, but it */
  /* seems a bit more compact and concise */
  for (work_atom = molecule_base; work_atom;
     work_atom = work_atom->next) {
  assign_q_s_descriptor(work_atom);
  }


  /* Initialize the lists */
  frag_list = NULL;
  atom_map_list = bond_map_list = NULL;
  dihedral_list = NULL;

  /* Open database communication, as we'll be needing it for awhile */
  socket_handle = get_new_socket(query_server_host, query_server_port);

  /* Begin checking all atoms in the molecule */

#ifdef EXTRA_OUTPUT
  {
  int query_count_guess = 0, i;

  for (work_atom = molecule_base; work_atom;
     work_atom = work_atom->next) {
    query_count_guess++;
  }
  query_count_guess++;  /*for number of atoms */
  fprintf(message_file, "Beginning %d atom match queries.  Each dot "
      "represents 5 atoms.\n", query_count_guess);
  fprintf(message_file, "                  \\");
  for (i = 0; i < query_count_guess; i++) {
    if(i % 5 == 0 && i != 0) {
    fprintf(message_file, ".");
    }
  }
  fprintf(message_file, "/\n                  \\");
  };
#endif

  for ( work_atom = molecule_base; work_atom; work_atom = work_atom->next ) {

  /* Local declarations */
  dihed my_dihedral;
  qdb_result work_result;
  boolean do_loop;
  atom *new_fragment;

#ifdef EXTRA_OUTPUT
  {
    if(get_atom_offset(work_atom) % 5 == 0 && work_atom != molecule_base) {
    fprintf(message_file, ".");
    fflush(message_file);
    }
  }
#endif

  /* Clean this up every time through, as it's thorougly re-used */
  BLANK_QDB_RESULT(work_result);

  my_dihedral = is_qcode_match_in_molecule_list(work_atom, NULL,
                        frag_list, tolerance);

  if (my_dihedral.list_index != -1) {
    /* We found a match in our own list, record it and move on */

    /* This section will need to copy my_dihedral.atom2_offset in */
    /* the bond matching section */
    work_result.atom1 = my_dihedral.atom1_offset;
    work_result.s_atom1 = get_atom_offset(work_atom);

    /* Store the list index in offset of work_result */
    work_result.offset = my_dihedral.list_index;

    atom_map_list = add_dbresult_to_qdb_result_list(atom_map_list, 1,
                          work_result);

    /* And, since we found an answer, we can skip the rest of this */
    /* loop */
    continue;
  }

  /* If we got this far, we need to check in the database for a */
  /* match */
  work_result = is_in_qdb(work_atom, NULL, tolerance, socket_handle);

  if ( work_result.atom1 != -1 ) {
    /* We found a result in the database!  Add it to atom_map_list */
    work_result.s_atom1 = get_atom_offset(work_atom);

    atom_map_list = add_dbresult_to_qdb_result_list(atom_map_list, 2,
                          work_result);

    continue;
  }

  /* Finally, if we're here, we need to generate a fragment to use. */
  /* This section is heavily copied from the old implementation of */
  /* this action */
  i = 0; /* i will be the fragment size */
```

```
  do_loop = true;
  while (do_loop) {
    new_fragment = generate_fragment(work_atom, NULL, i);
    if (is_qcode_match(work_atom, new_fragment, tolerance)) {
    /* We've got a match!  Make sure to record it's location */
    /* in our match list, so later programs know what we */
    /* need */

    frag_list = add_to_molecule_list(new_fragment, frag_list);

    /* Since our fragment is in the list now (we put it there), */
    /* borrow from the first if statement on how to remember */
    /* where it is. */
    my_dihedral = is_qcode_match_in_molecule_list(work_atom, NULL,
                          frag_list, tolerance);

    work_result.atom1 = my_dihedral.atom1_offset;
    work_result.s_atom1 = get_atom_offset(work_atom);

    /* Store the list index in offset of work_result */
    work_result.offset = my_dihedral.list_index;

    atom_map_list = add_dbresult_to_qdb_result_list(atom_map_list, 1,
                          work_result);

    do_loop = false;
    } else {
    i++;
    free_molecule(new_fragment);
    }
  }
  }

#ifdef EXTRA_OUTPUT
  {
  int query_count_guess = 0;
  fprintf(message_file, "/\n");

  for (work_atom = molecule_base; work_atom;
     work_atom = work_atom->next) {
    query_count_guess++;
  }
  query_count_guess++;  /*for number of atoms */
  fprintf(message_file, "Begin bond queries:");
  fprintf(message_file, "  \\");
  };
#endif

  /* Begin checking all bonds in the molecule.  This section is a bit */
  /* more complicated.  The 'ideal' algorithm is probably recursive, */
  /* but for simplicity, we'll simply loop through and use mark2() on */
  /* the bonds we've visited to keep track of them */

  /* Ok, a word of warning to the wise, if a function uses state (and */
  /* modifies it) it _must_ make a copy of them before changing them, */
  /* (i.e., using recurse_molecule_do), and restore them when it's done */
  /* with them!  (This particular problem stymied me for quite some time) */

  /* Before we start, make sure all of the states are zeroed */
  molecule_zero_states(molecule_base);

  for ( work_atom = molecule_base; work_atom; work_atom = work_atom->next ) {

#ifdef EXTRA_OUTPUT
  {
    if(get_atom_offset(work_atom) % 5 == 0 && work_atom != molecule_base) {
    fprintf(message_file, ".");
    fflush(message_file);
    }
  }
#endif

  /* First things first, we need to find an un-visited bond on this */
  /* atom */
  for ( i = 0; i < work_atom->valence; i++ ) {

    /* Local declarations */
    dihed my_dihedral;
    qdb_result work_result;
    boolean do_loop, is_match;
    atom *new_fragment, *other_end;

    if ( check2(work_atom->state, i)  ) {
    /* We've been here before, go to the next iteration */
    continue;
    }

    /* We can only get here if we're on a valid bond, so here we go */
    /* Mark the bit on both atoms */
    mark2( &(work_atom->state), i);
    /* Find the remote bond that points back here */
    for ( j = 0; j < work_atom->valence; j++ ) {
    other_end = work_atom->bond[i];
    if (other_end->bond[j] == work_atom) {
      /* We found it! */
      mark2( &(other_end->state), j );
      j = work_atom->valence;
    }
    }

    /* Finally, we can get to looking for this beast */

    /* Clean this up every time through, as it's thorougly re-used */
    BLANK_QDB_RESULT(work_result);

    my_dihedral = is_qcode_match_in_molecule_list(work_atom,
                      other_end, frag_list, tolerance);

    if (my_dihedral.list_index != -1) {

    /* We may have found a match in our own list, record it */
    /* and move on */

    work_result.atom1 = my_dihedral.atom1_offset;
```

```
work_result.atom2 = my_dihedral.atom2_offset;
work_result.s_atom1 = get_atom_offset(work_atom);
work_result.s_atom2 = get_atom_offset(other_end);

/* Store the list index in offset of work_result */
work_result.offset = my_dihedral.list_index;

/* And tolerance2 is not used in the next function call, */
/* make it a 'bad' value so errors are caught. */
work_result.tolerance2 = -1.0;

/* Verify asymmetry in environment */
j = compare_asymmetry_environment_atom(
    work_result, molecule_base, frag_list,
    asymmetry_range, tolerance);

/* Since tolerance2 is unused in work_result, use it to */
/* mark what kind of match we have if we actually got any */
if ( j ) {    /* <---   == 1 or 2 */
  work_result.tolerance2 = j;

  /* Save this match */
  bond_map_list = add_dbresult_to_qdb_result_list(
            bond_map_list, 1, work_result);
  /* And, since we found an answer, we can skip the rest */
  /* of this loop.  */
  continue;
} else {
  /* The local environments are diastereotopically */
  /* realated, or have different numbers of stereogenic */
  /* carbon's in range.  So we simply fall out of the */
  /* loop and keep looking */
  BLANK_QDB_RESULT(work_result);
}
}

/* If we got this far, we need to check in the database for a */
/* match */
work_result = is_in_qdb(work_atom, other_end, tolerance,
            socket_handle);

if ( work_result.atom1 != -1 ) {
boolean local_match;

/* We found a result in the database!  Before we can */
/* actually add it to our bond_map_list, however, */
/* we need to verify that any asymmetry in the */
/* molecule is accounted for in the database's */
/* entry */
work_result.s_atom1 = get_atom_offset(work_atom);
work_result.s_atom2 = get_atom_offset(other_end);

local_match = false;

j = compare_asymmetry_environment_db(work_result,
    molecule_base, socket_handle, asymmetry_range, tolerance);

/* Since tolerance2 is unused in work_result, use it to */
/* mark what kind of match we have if we actually got any */
if ( j ) {    /* <---   == 1 or 2 */
  work_result.tolerance2 = j;
  /* Save this match */
  bond_map_list = add_dbresult_to_qdb_result_list(
                  bond_map_list, 2, work_result);

  /* And, since we found an answer, we can skip the rest */
  /* of this loop.  */
  continue;
} else {
  /* The local environments are diastereotopically */
  /* realated, or have different numbers of stereogenic */
  /* carbon's in range.  So we simply fall out of the */
  /* loop and keep looking */
  BLANK_QDB_RESULT(work_result);
}
}

/* Finally, if we're here, we need to generate a fragment to use. */
/* This section is heavily copied from the old implementation of */
/* this action */
j = 1; /* j will be the fragment size */
do_loop = true;
while (do_loop) {
atom *new_other_end;

is_match = false;

new_fragment = generate_fragment(work_atom, other_end, j);
if (is_qcode_match(work_atom, new_fragment, tolerance)) {
  /* We've got one side of a match, now match the other */
  /* side as well */
  for ( k = 0; k < new_fragment->valence; k++) {
  if ( is_qcode_match(other_end, new_fragment->bond[k],
            tolerance ) ) {
    /* We found our match, record this atom and */
    /* move on */
    is_match = true;
    new_other_end = new_fragment->bond[k];
    k = new_fragment->valence;
  }
  }

  if ( is_match == false ) {
  /* Increment j, and move on. */
  free_molecule(new_fragment);
  j++;
  continue;
  }
  /* Now, we definately have a match on both sides, and */
  /* we even know where it is, so let's record the */
  /* information and add it to our bond match list */

  frag_list = add_to_molecule_list(new_fragment, frag_list);

    /* Since our fragment is in the list now (we put it there),*/
    /* borrow from the first if statement on how to remember*/
    /* where it is. */
    my_dihedral = is_qcode_match_in_molecule_list(work_atom,
                      other_end, frag_list, tolerance);

    work_result.atom1 = my_dihedral.atom1_offset;
    work_result.atom2 = my_dihedral.atom2_offset;
    work_result.s_atom1 = get_atom_offset(work_atom);
    work_result.s_atom2 = get_atom_offset(other_end);

    /* And ... since we know the new fragment must be homomeric */
    /* since we just made it from a template */
    work_result.tolerance2 = 1;

    /* Store the list index in offset of work_result */
    work_result.offset = my_dihedral.list_index;

    bond_map_list = add_dbresult_to_qdb_result_list(
                        bond_map_list, 1, work_result);

    do_loop = false;
  } else {
    free_molecule(new_fragment);
    j++;
  }
  }
}
}

#ifdef EXTRA_OUTPUT
  {
  fprintf(message_file, "/\n");
  }
#endif

  /* FINISHED */

  /* The previous section appears to work fine, though there may still */
  /* be multitudes of logical errors.  The qdb_searching has been */
  /* reasonably well tested, and the generation of fragments has seen */
  /* a fair bit of life in previous incarnations of this program.  The */
  /* first several 'production runs' need to watch it carefully */

  /* Ok, before we do the output, we need to enumerate the meanings of */
  /* the qdb_result's in the lists.  The members and thier meanings are: */
  /* directory  -> Directory name iff source was qdb (via a query) */
  /* atom1    -> Atoms matched in actual fragments to be calculated */
  /* atom2       or have data extracted from.  If atom2 == -1, it */
  /*              was an atom match */
  /* tolerance1  -> The source of the data, it should be 1 if the data */
  /*          comes from the frag_list, and 2 if the data comes */
  /*              from the quantum chemistry database */
  /* tolerance2  -> The 'sense' of any asymmetry match, -1 if there is */
  /*              no asymetry in the neighborhood to be concerned */
  /*              about, 0 on a failed call to any of the functions */
  /*              that report on asymmetry, 1 if the relationship is */
  /*              homomeric, and 2 if it's enantiomeric */
  /* s_atom1    -> Atoms matched in the parent molecule, if s_atom2 is */
  /* s_atom2        -1, it was an atom match */
  /* offset    -> Either -1 (if unused) or the position in the frag_list */

  /* Begin output for next program in the pipe */

  out_file = stdout;

  /* The next program would probably like to know what the parent molecule */
  /* is.  By outputting it's structure here, the data remains independent */

  fprintf(out_file, "Begin parent molecule:\n");

  /* Print coordinates */
  fprintf(out_file, "Begin coordinates\n");
  for (work_atom = molecule_base - get_atom_offset(molecule_base);
    work_atom; work_atom = work_atom->next) {
  fprintf(out_file, "%s,%g,%g,%g\n",
     work_atom->label,
     work_atom->coordinates[0],
     work_atom->coordinates[1],
     work_atom->coordinates[2] ) ;
  }

  /* Print connectivity */
  fprintf(out_file, "Begin connectivity\n");
  print_molecule_connectivity(out_file, molecule_base);

  /* Print out the qcodes */
  fprintf(out_file, "Begin qcodes\n");
  for (work_atom = molecule_base - get_atom_offset(molecule_base);
    work_atom; work_atom = work_atom->next) {
  for (j = 0; j < QDEPTH; j++ ) {
    if ( j ) { fprintf(out_file, " "); }
    /* Note the precision specifier .255.  It is */
    /* important to catch every last digit for recording */
    /* in the Qcodes files. */
    fprintf(out_file, "%.255Lg", work_atom->qcode[j]);
  }
  fprintf(out_file, "\n");
  }

  /* And finally, print any stereochemical descriptors that may */
  /* need to be printed.  We'll use the list provided by */
  /* atom_list_manage() for this */

  atom_list_manage(NULL, A_CLEAR);
  for (work_atom = molecule_base - get_atom_offset(molecule_base);
    work_atom; work_atom = work_atom->next) {
  if ( is_asymmetric_carbon(work_atom) ) {
    atom_list_manage(work_atom, A_PUSH);
  }
  }
  work_atom = molecule_base - get_atom_offset(molecule_base);
  if ( atom_list_manage(work_atom, A_COUNT) != work_atom ) {
  /* We have atoms to report */
```

```
fprintf(out_file, "Begin stereochemical descriptors\n");
while ( (work_atom = atom_list_manage(NULL, A_POP) )
    != NULL ) {
  fprintf(out_file, "%d %c\n",
      get_atom_offset(work_atom),
      work_atom->s_descriptor );
}
}

fprintf(out_file, "End molecule output\n");

/* These two lists print out the information needed by the next program */
/* concerning which parent atoms and bonds correspond to which atoms */
/* and bonds in the fragments, and where the fragments can be found */
fprintf(out_file, "Begin atom map list:\n");
print_result_list(out_file, atom_map_list);
fprintf(out_file, "End atom map list:\n");

fprintf(out_file, "Begin bond map list:\n");
print_result_list(out_file, bond_map_list);
fprintf(out_file, "End bond map list:\n");

if ( frag_list == NULL ) {
fprintf(out_file, "No new fragments for the database\n");
} else {

/* We need to output the information on the new fragments */

int j;

fprintf(out_file, "Begin new fragments:\n");
/* Begin printing out fragments */
for ( i = 0; frag_list[i]; i++ ) {
  fprintf(out_file, "Fragment list molecule number %i\n", i);

  /* Print coordinates */
  fprintf(out_file, "Begin coordinates\n");
  for (work_atom = frag_list[i] - get_atom_offset(frag_list[i]);
     work_atom; work_atom = work_atom->next) {
  fprintf(out_file, "%s,%g,%g,%g\n",
      work_atom->label,
      work_atom->coordinates[0],
      work_atom->coordinates[1],
      work_atom->coordinates[2] ) ;
  }

  /* Print connectivity */
  fprintf(out_file, "Begin connectivity\n");
  print_molecule_connectivity(out_file, frag_list[i]);

  /* Print out the qcodes */
  fprintf(out_file, "Begin qcodes\n");
  for (work_atom = frag_list[i] - get_atom_offset(frag_list[i]);
     work_atom; work_atom = work_atom->next) {
  for (j = 0; j < QDEPTH; j++ ) {
    if ( j ) { fprintf(out_file, " "); }
    /* Note the precision specifier .255.  It is */
    /* important to catch every last digit for recording */
    /* in the Qcodes files. */
    fprintf(out_file, "%.255Lg", work_atom->qcode[j]);
  }
  fprintf(out_file, "\n");
  }

  /* And finally, print any stereochemical descriptors that may */
  /* need to be printed.  We'll use the list provided by */
  /* atom_list_manage() for this */

  atom_list_manage(NULL, A_CLEAR);
  for (work_atom = frag_list[i] - get_atom_offset(frag_list[i]);
     work_atom; work_atom = work_atom->next) {
  if ( is_asymmetric_carbon(work_atom) ) {
    atom_list_manage(work_atom, A_PUSH);
  }
  }
  work_atom = frag_list[i] - get_atom_offset(frag_list[i]);
  if ( atom_list_manage(work_atom, A_COUNT) != work_atom ) {
  /* We have atoms to report */
  fprintf(out_file, "Begin stereochemical descriptors\n");
  while ( (work_atom = atom_list_manage(NULL, A_POP) )
      != NULL ) {
    fprintf(out_file, "%d %c\n",
        get_atom_offset(work_atom),
        work_atom->s_descriptor );
  }
  }

  /* And ... end this fragment's description */
  fprintf(out_file, "End molecule output\n");
}
fprintf(out_file, "Output complete\n");
}

/* Uncomment the following section to get output files that can be */
/* looked at with gaussview for each of the fragments, and feel free */
/* To change the destination directory *grin* */
/* if ( frag_list ) { */
/*   for(i = 0; frag_list[i]; i++) { */
/*     sprintf(work_string, "/home/radke/frags/frag%i.com", i); */
/*     printf("Will try to create file:%s\n", work_string); */
/*     tmp_file = fopen(work_string, "w"); */
/*     if ( !tmp_file ) { */
/*       error_exit("Unable to open new file for writing, *boggle"); */
/*     } */
/*     atom_print_com(tmp_file, frag_list[i]); */
/*     fclose(tmp_file); */
/*   } */
/* } else { */
/*   printf("Frag list is empty, not saving frag files\n"); */
/* } */


/* A note on performance.  When the program needs to calculate */
/* connectivity for itself, it calls get_bond_order for every combination */
  /* of atoms.  This results in not much of a perfomance penalty, since */
  /* the real bottleneck is the socket communication (i.e., on a 20 */
  /* second run, the program is only running 70 milliseconds total */

  /* Aside from several bits allocated by the socket library, the program */
  /* with the following free calls leaks exactly 0 memory */
  if ( frag_list ) {
  for (i = 0; frag_list[i]; i++) {
    free_molecule(frag_list[i]);
  }
  free(frag_list);
  }

  if ( atom_map_list ) {
  for (i = 0; atom_map_list[i]; i++) {
    free(atom_map_list[i]);
  }
  free(atom_map_list);
  }

  if ( bond_map_list ) {
  for (i = 0; bond_map_list[i]; i++) {
    free(bond_map_list[i]);
  }
  free(bond_map_list);
  }

  /* Prepare_asymmetry_query returns a newly allocated string, and even */
  /* though it re_uses the same space (self_cleans).  A call to it with */
  /* asymmetry_range == -999 results in it cleaning it's memory and */
  /* returning */
  prepare_asymmetry_query(work_result, NULL, 0.0, -999);

  free_molecule(molecule_base);
  close(socket_handle);

#ifdef DMALLOC
  dmalloc_shutdown();
#endif

  return 1;
}
```

generate_fragment.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include "fraggen.h"

/* This function generates a fragment of the original molecule to */
/* whatever depth is specified in the function call.  Finally, it */
/* returns a pointer to the base of the array that the new molecule */
/* lives at.  Please note that this function allocates memory, and */
/* the memory _MUST_ be freed later to prevent memory leaks */
/* Ok, not so finally.  After the original version was working, it was */
/* decided that this function should handle the general case of either */
/* an atom centered, or a bond centered fragment.  In order to get an */
/* atom centered fragment, simply call it with second_atom = NULL.  To */
/* get a bond centered fragment, call it with a second atom.  Note that */
/* the function explicityly checks to make certain they are bonded, since */
/* a fragment generated with distant atoms is meaningless.  Ok, after */
/* coming back to this function after a bit of time away, it became */
/* apparent that it is not necessarily very well written (and, it seems */
/* to have broken).  It will be re-written with a similar algorithm, but */
/* somewhat cleaner implementation, copying it's style */
/* from recurse_molecule_do() */
atom *generate_fragment(atom *p_atom1, atom *p_atom2, int depth) {

atom *new_fragment, *work_atom, *new_fragment_base, *old_molecule_base;
atom **dangle_list, **frag_members;
long int *old_states;
int i, j,old_size;
boolean do_loop, is_bond;

/* Let's do some error checking */
if ( p_atom1 == NULL ) {
  warn_out("NULL pointer passed to generate_fragment(), this was likely "
       "unintended, and may be fatal");
}

if ( p_atom2 != NULL ) {
  if ( ! is_bonded(p_atom1, p_atom2) ) {
    warn_out("non-bonded second atom passed to generate_fragment, this "
         "was likely unintended, but should not be fatal (though"
         "the results could be fascinating *grin*");
  }
}
```

```
if ( depth < 0 ) {
    warn_out("Cannot make fragment of size less than 0 in "
      "generate_fragment(), returning NULL");
    return NULL;
}

/* Before we start, here is the general algorithm: */
/* 1) Duplicate the parent molecule, this space is eventually where the */
/*    entire new fragment will be */
/* 2) Mark the states of atoms 'in range', via a recursive algorithm very */
/*    similar to recurse_molecule_do */
/* 3) If we're building a bond centered fragment, save the states of the */
/*    new molecule, do the same recursive access, and combine the states */
/*    from the first pass with the states from this pass */
/* 4) Initialize the dangle list (Atoms that may need to be turned onto */
/*    Hydrogen's with appropriate lengths)  */
/* 5) Initialize ( on the same pass as #4) a list of all of the atoms */
/*    that were 'properly' part of the fragment */
/* 6) Until no members of the dangle list have bonds that point to other */
/*    members of the dangle list, move dangle list members to the */
/*    frag_members list, add all non frag_members to the dangle list, */
/*    and re-mark states of new dangle members to the same state that */
/*    the new frag_member had.  This step is quite tedious, but will */
/*    prevent fragmenting rings that are nearly included in the fragment */
/* 7) Break all bonds and remove portions of the molecule given by */
/*    members of the dangle list, and not along bonds pointing to */
/*    members of the frag_members list */
/* 8) Change all dangle list members into H's, and correct thier bond */
/*    lengths */
/* 9) Return new fragment */


/* Initializations */
dangle_list = frag_members = NULL;
dangle_list = add_to_molecule_list(NULL, dangle_list);
frag_members = add_to_molecule_list(NULL, frag_members);
old_molecule_base = p_atom1 - get_atom_offset(p_atom1);

/* Fast forward to the end */
for (work_atom = p_atom1; work_atom->next;
  work_atom = work_atom->next ) {}
old_size = get_atom_offset(work_atom) + 1;

/* Are we processing a bond? */
is_bond = ( p_atom2 == NULL ) ? false : true;

/* Get our new fragment */
new_fragment = duplicate_molecule(p_atom1);
new_fragment_base = new_fragment - get_atom_offset(p_atom1);

/* Mark it */
mark_recurse_fragment( new_fragment, depth );

if (is_bond) {

  long int *old_states;
  atom *new_frag_other_end, *work_atom;

  /* Warning, save_states allocates new memory, and it must be freed */
  /* Remember!  The states returned from save_states are 1 based, */
  /* with the 0 holding the list index! */
  old_states = save_states( new_fragment );
  new_frag_other_end = new_fragment_base + get_atom_offset(p_atom2);

  /* Mark the atom starting from the other end of the bond */
  mark_recurse_fragment( new_frag_other_end, depth );

  work_atom = new_fragment_base;

  /* Combine the states from the two runs */
  for ( i = 0; i < old_size; i++ ) {
    if ( work_atom[i].state == LONG_MAX &&
      old_states[i + 1] == LONG_MAX) {
      continue;  /* Do nothing */
    }

    /* Combine the states */
    if ( old_states[i + 1] < work_atom[i].state ) {
      work_atom[i].state = old_states[i + 1];
    }
  }
  free(old_states);
}

/* Ok, we will not process unmarked atoms, and for the marked */
/* ones, we have several tasks. We need to add all of the atoms */
/* we see here to our frag_members list.  We need to add all */
/* unmarked atoms connected to atoms whose states == depth to our */
/* dangle list.  Note that in the else section that follows, */
/* we do all of these except the combining states lists */

work_atom = new_fragment_base;

/* Add the appropriate atoms to their lists */
for ( i = 0; i < old_size; i++ ) {

    if ( work_atom[i].state == LONG_MAX ) {
      continue;  /* Do nothing */
    }

  /* Add this atom to our frag_members list */
  frag_members = add_to_molecule_list( work_atom + i, frag_members);

  /* If we're in the right place, add un-member atoms to dangle */
  /* list */
  if ( work_atom[i].state == depth ) {
    for (j = 0; j < work_atom[i].valence; j++) {
      if ( work_atom[i].bond[j]->state == LONG_MAX) {
        dangle_list = add_to_molecule_list(work_atom[i].bond[j],
          dangle_list);
      }
    }
  }
}
```

```
/* New molecule is now fully marked, and we move on to step #6 */
/* Note:  This section is completely untested, aside from syntax!!! */
/* None of the molecules I'm working with have strange functionalities */
/* like unconjugated rings (especially epoxides).  When that time comes, */
/* it may well be very buggy!!!!  On second though, I'm not even going */
/* to write the code now, I'll simply leave a stub */

while (do_loop) {
  /* Declarations */
  int k;

  /* Initializations */
  do_loop = false;

  /* Processing */
  for ( i = 0; dangle_list[i]; i++ ) {
    for ( j = 0; j < dangle_list[i]->valence; j++ ) {
      for (k = 0; dangle_list[k]; k++) {
        if ( dangle_list[i]->bond[j] == dangle_list[k] ) {
          /* This is the point where the tricky bit of */
          /* algorithm step #6 needs to be implemented */
          error_exit("Members bonded to each other found "
                   "in dangle_list in generate_fragment().  "
                 "This is a special condition that needs "
                 "to be handled, and this exit is a stub "
                 "for that task");
          do_loop = true;
        }
      }
    }
  }
}


/* And on to step #7 */

/* Since delete_molecule_group leaves the states untouched, we need to */
/* set them ourselves, but we need to save the old states and use _them_ */
/* for comparisons to decide if we're going to be leaving */
old_states = save_states(new_fragment);

molecule_max_states(new_fragment);

for ( i = 0; dangle_list[i]; i++ ) {
  for (j = 0; j < dangle_list[i]->valence; j++) {
    if ( old_states[
      get_atom_offset(dangle_list[i]->bond[j]) + 1
    ] == LONG_MAX) {
      delete_molecule_group( dangle_list[i], j);
    }
  }
}

free(old_states);
new_fragment = repack_molecule(new_fragment);
new_fragment_base = new_fragment;

/* Finally, in the (almost) last step, we find the atoms in our new */
/* new fragment who do not have full valences, and turn them into */
/* hydrogens, while resetting the bond lengths to 'better values */
for (work_atom = new_fragment; work_atom; work_atom = work_atom->next ) {
  if ( !is_atom_valence_full(work_atom) ) {
    float x, y;
    vector_d work_vector, work_vector2;

    strcpy(work_atom->label, "H");
    work_atom->atomic_number = 1;

    /* For those who haven't played with linear algebra for awhile, */
    /* the next section can look a bit like wizardry ... I don't know */
    /* if there's an easy solution to this or not */
    x = get_single_bond_length(1, work_atom->bond[0]->atomic_number);
    work_vector = vec_subtract(work_atom->coordinates,
                (work_atom->bond[0])->coordinates, 3);
    y = vec_scalar_length(work_vector, 3);
    work_vector2 = vec_scalar_multiply( x/y, work_vector, 3 );

    /* Remember, many of these vectors are allocated, and must */
    /* be freed after getting them from vec_ functions.  See */
    /* ../general/vector.h for a list of which allocate new memory */

    /* Now that we have the new vector, copy it in */
    free(work_atom->coordinates);
    work_atom->coordinates = vec_add(work_vector2,
      (work_atom->bond[0])->coordinates, 3);

    /* And ... clean up */
    free(work_vector);
    free(work_vector2);

  }
}

#ifdef DMALLOC
dmalloc_verify(0);
#endif

free(frag_members);
free(dangle_list);

/* Finally, before we return, we need to re-generate the qcodes */
if ( assign_qcodes(new_fragment) == 0 ) {
    warn_out("Failed to initialize all qcodes in generate_fragment()");
}

/* And ... free the memory */
mark_recurse_fragment ( NULL, 0 );

/* Also, it is most convenient for the calling environment if the new */
/* fragment we returned points to the atom we build the fragment around. */
/* Unfortunately, we don't know which it is, so we'll search the new */
/* fragment for another atom with the same (exact) coordinates */
for(work_atom = new_fragment_base; work_atom;
```

```
  work_atom = work_atom->next) {
  if ( work_atom->coordinates[0] == p_atom1->coordinates[0] &&
     work_atom->coordinates[1] == p_atom1->coordinates[1] &&
     work_atom->coordinates[2] == p_atom1->coordinates[2] ) {
    return work_atom;
  }
}

/* If we reach here, it means we lost the staring atom somehow, and */
/* need to indicate that */
warn_out("Lost starting atom in generate_fragment(), returning NULL");
free(new_fragment);
return NULL;
}

/* The following variable needs to be global for the recurse functions */
/* to work */
static int max_recurse_depth;

/* The following function is a wrapper for the function of the same name */
/* ( with _core appended ).  It is largely copied from atom_handling.c:: */
/* recurse_molecule_do().  The main difference in how it works is that */
/* it is resposible for _not_ incrementing the state count ( on the next */
/* atom ) if we're about to travel over a bond whose return value from */
/* isin_group() is true.  The difference between this recursion and */
/* recurse_molecule_do() is that the state will not necessarily be */
/* decremented on each visit to the fucntion.  In order to prevent */
/* infinite recursion (where it simply goes back and forth along one */
/* bond), we also need to pass the source atom to the core, and prohibit */
/* it from returning to that atom.  If NULL is passed as center, it */
/* simply frees the memory in old_states, and exits */
long int *mark_recurse_fragment( atom *center, int depth ) {

static long int* old_states = NULL;

/* Free the memory held by old_states if it has ever been allocated */
if (old_states) {
  free(old_states);
  old_states = NULL;
}

if ( center == NULL ) {
  return NULL;
}

old_states = save_states(center);

molecule_max_states(center);

max_recurse_depth = depth;

mark_recurse_fragment_core( center, center, depth );

return old_states;
}

/* This is the core of the above function.  See the comments for the */
/* wrapper for more information on how this function works */
void mark_recurse_fragment_core( atom* this_atom, atom *source, int depth ) {

/* I'm not sure if declaring this static results in any speed increase, */
/* but the idea is that the function doesn't really need to allocate */
/* a new one every time it visits, as re-using the old one should be */
/* just fine (since we always initialize it */
static int this_state;

/* Initialization */
this_state = max_recurse_depth - depth;

/* End the recursion if we're deeper than another visit here.  This */
/* test prevents re-marking when we come around a ring, for example */
if ( this_state >= this_atom->state || depth < 0 ) { return; }

/* Prepare for next call */
this_atom->state = this_state;

/* And check the bonds */
{
  register int i, i_v;
  atom *next_atom;
  i_v = this_atom->valence;
  for ( i = 0; i < i_v; i++ ) {
    next_atom = this_atom->bond[i];
    /* Only decrement the depth if we're not going down an */
    /* isin_group() bond (and not returning to a previous atom */
    if ( next_atom != source ) {
      if ( isin_group(this_atom, i) ) {
        mark_recurse_fragment_core( this_atom->bond[i],
          this_atom, depth );
      } else {
        mark_recurse_fragment_core( this_atom->bond[i],
          this_atom, depth - 1 );
      }
    }
  }
}

return;
}

/* The following function is a 'helper' function to the main */
/* generate_fragment function.  It can be expanded to include arbitrary */
/* rules, or eventually to use a configuration file.  For now, it will */
/* simply consider a group something that is connected by a bond order */
/* greater than 1.0.  */
boolean isin_group(atom *this_atom, int bond_index) {

/* This is the only rule right now.  Other rules can be added here */
if (this_atom->bond_order[bond_index] > 1.0) {
  return yes;
}

return no;
}
```

assign_qcodes.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include <math.h>
#include "../general/atom.h"

/* QDEPTH is the number of doubles in the final qcode */
#ifndef QDEPTH
#define QDEPTH 20
#endif

#ifndef MAXSTR
#define MAXSTR 255
#endif

/* This routine assigns qcodes to all of the atoms in the molecule */
/* specified by *some_atom.  Note that the molecule must be properly */
/* defined by the atom->previous and atom->next pointers.  Note also */
/* that some of the qcode vector components are different in the 6th */
/* decimal place.  This is most likely because it was compiled with the */
/* cc -fast option, though this has not been verified.  Note also that */
/* ../general/atom_handling.c:atom_get_pauling_elecneg() has had a */
/* couple of changes, see the note there for documentation */
int assign_qcodes(atom *some_atom) {

atom *head, *work_atom;
int i, j;
long double sum;
boolean leave_routine;
char work_string[MAXSTR];

/* Find head of molecule */
head = some_atom - get_atom_offset(some_atom);

/* First, do some error checking.  If any of the atoms in the molecule */
/* have a 0 valence, it will crash the routine, and this routine should */
/* not be used with unfinished atoms at any rate */
work_atom = head;
leave_routine = no;
while (work_atom != NULL) {
  if (work_atom->valence == 0) {
    sprintf(work_string, "zero valence atom at offset %d passed to "
              "assign_qcodes()", get_atom_offset(work_atom) );
    warn_out(work_string);
    leave_routine = yes;
  }
  work_atom = work_atom->next;
}
if (leave_routine == yes) { return 0; }

/* Allocate enough space for QDEPTH + 1 doubles in the qcode pointer of */
/* each atom.  qcode[0] of each atom will be reserved for the normalizing */
/* electronegativity, or EN[][1] in the ../log2str/charge_map.c */
/* implementation of this algorithm. */

work_atom = head;
while (work_atom != NULL) {
  if ( ( work_atom->qcode =
  malloc( (QDEPTH + 1) * sizeof(long double) ) ) == NULL ) {
    warn_out("not enough memory to initialize a qcode vector"
      "in assign_qcodes");
    return 0;
  }
  work_atom = work_atom->next;
}

/* This is the tedious part.  Assign pauling electronegativities to each */
/* atom in the molecule.  Then, re-assign the reduced electronegativities */
work_atom = head;
while (work_atom != NULL) {
  work_atom->qcode[0] =
    (long double)atom_get_pauling_elecneg(work_atom->atomic_number);
  if (work_atom->qcode[0] == 0) {
    warn_out("electronegativity not available in assign_qcodes");
  }
  work_atom->qcode[0] = work_atom->qcode[0] /
    sqrt( (long double)(1 + atom_get_number_of_bonds(work_atom) ) );
  work_atom = work_atom->next;
}

/* With the reduced and normalized electronegativities calculated and */
/* in place, we now do the first part of the qcode algorithm, which is */
/* generating the EN vector which will be later translated into the */
/* qcode vector.  The formula by which we do this is: */
/* EN(i) = (average(EN(i-1)'s of neighbors) + EN(1)) / 2 */
```

```
/* Recall that EN(1) is stored in atom->qcode[0] */
for(i = 1; i <= QDEPTH; i++) {
  work_atom = head;
  while (work_atom != NULL) {
    sum = 0;
    for (j = 0; j < work_atom->valence; j++) {
      sum += work_atom->bond[j]->qcode[i - 1];
    }
    work_atom->qcode[i]=(sum/work_atom->valence + work_atom->qcode[0])/2;
    work_atom = work_atom->next;
  }
}

/* The last step of the algorithm is to assign the qcodes according to the */
/* formula Q(i) = ( EN(i) - EN(1) ) / EN(1).  Recall that EN(1) is stored */
/* in atom->qcode[0].  Note that the loop nesting is revers of the prvious */
/* operation.  This is not necessary, but it is the intuitively right */
/* thing, since we are simply updating each Qcode vector to give the final */
/* product */

work_atom = head;
while (work_atom != NULL) {
  for(i = 1; i <= QDEPTH; i++) {
    work_atom->qcode[i] = work_atom->qcode[i] / work_atom->qcode[0] - 1;
  }
  work_atom = work_atom->next;
}


/* Finally, repack the vectors, discarding atom->qcode[0] */
work_atom = head;
while (work_atom != NULL) {
  for(i = 1; i <= QDEPTH; i++) {
    work_atom->qcode[i - 1] = work_atom->qcode[i];
  }
  if (( work_atom->qcode = realloc(work_atom->qcode,
    QDEPTH * sizeof(long double))) == NULL ) {
    warn_out("not enough memory to realloc a qcode vector"
      " in assign_qcodes");
    return 0;
  }
  work_atom = work_atom->next;
}

return 1;
}
```

Miscellaneous Programs

configure.pl

```
#!/usr/bin/perl -wT

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

package main;

eval { require 5.6.1 }
  or die <<MESSAGE;
##################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
##################################################################
MESSAGE

use strict;

# Before proceeding, clean up our environment so we can run external
# programs
require '../general/clean_environment.pl';
full_env_clean();

# This script will configure the makefile for whatever operating system
# we are running on.  It gets this information from the environment
# variable $OSTYPE, which will be laundered and used immediately as
# if it was secure.  I cannot perceive of any circumstances that would
# make this a security problem.

my($os)=$^O;

if ($os =~ /^([-\@\w.]+)$/) {
  $os = $1;  #$os is laundered
} else {
  die "Bad system \"$os\" retrived from environment, exiting";
}

if ( $os eq '' ) {
  die "This system does not have the environment variable OSTYPE " .
    "defined.  Please define it so your makefile can be configured";
}

my($path_to_root) = '../';

open ("MAKEFILE", ">./Makefile") or die
  "Unable to open makefile for writing, exiting";

print MAKEFILE <<MY_MESSAGE
######################################################################
# This makefile was created by configure.pl.  Any changes to this
# makefile will be overwritten!  Please edit the configure.pl script
# instead.
#
#
MY_MESSAGE
;

# Values for files in this package, these are the mostly likely to
# change regularly, so they're put first
my($vpath) = "../log2str:../general";
my($mainobjs) = "qdb_check.o";
my($headers) = "\tqdb_check.h atom.h vector.h fraggen.h\\\n" .
        "\t\tqdb_shared_functions.h";
my($otherobjs) = "\tqdb_check_functions.o vector_handling.o\\\n" .
        "\t\tget_bond_order.o assign_qcodes.o\\\n" .
        "\t\tatom_handling.o generate_fragment.o\\\n" .
        "\t\tmy_socket.o qdb_shared_functions.o\n";
my($exefile) = "qdb_check";
my($miscdeps)=<<MORESTUFF
vector_handling.o: ../general/vector.h
atom_handling.o: ../general/atom.h get_bond_order.o vector_handling.o\\
  qdb_shared_functions.o
assign_qcodes.o: vector_handling.o atom_handling.o
generate_fragment.o: vector_handling.o atom_handling.o
qdb_check_functions.o: qdb_check.h vector_handling.o\\
  my_socket.o atom_handling.o qdb_shared_functions.o
qdb_check.h: vector_handling.o atom_handling.o fraggen.h
my_socket.o: ../general/my_socket.h qdb_shared_functions.o
MORESTUFF
;

# Default makefile values

# Note:  Just a reminder.  my($variable) variables are local only to this
# scope, which is main.  If another file is 'require' 'ed, it will _not_
# have access to these variables unless they're fully qualified (since
# I'm using strict vars).  Any variables that are changed in the os
# specific instructions must be scoped to main here.

$main::debug = "-g";
$main::profile = "";
$main::base = "";
$main::cc = "cc";
$main::cflags = '$(DEBUG) $(PROFILE)';
$main::libs = "-lm";
$main::incls = "";
$main::defines = "";
my($dmalloc) = "";
my($exeshellargs) = "";
my($nowarn) = "";

# Include any information for system specific options
my ($sys_config_file) = $path_to_root . 'general/os_specific/' .
  $os . '_make.pl';

if ( -r $sys_config_file ) {
  # Announce our finding.  Note, the file itself should provide the
  # newline, and any other additional information to print.
  print "Preparing makefile for $os.  ";

  # And use it
  require $sys_config_file;
} else {
  print "Don't know how to make makefile for $os (i.e., file\n" .
    "\"general/os_specific/${os}_make.pl\" not found).  Using \n" .
    "generic configuration file\n";
  require $path_to_root . 'general/os_specific/generic_make.pl';
}


# I haven't worked out anything too fancy for command line handling, but
# for now, I'll add the options in hodgepodge, and document as I go.  Note
# That command line processing doesn't happen until _after_ the defaults
# are entered.  This allows any of the defaults to be overwritten.  If any
# arguments that normally take a string are left empty, they will also
# be empty in the resulting Makefile.

# Special variables used only in dealing with command line options:
my($addcflags);
my($addlibs);
my($addincls);
my($adddefines);
my($help_flag);
my($profile_on);

my($all_options) = q/
  "debug|d:s"   =>  \$main::debug,
  "profile|p:s" =>  \$main::profile,     # Ignored unless -P is also provided
  "P"           =>  \$profile_on,  # Set this to turn profiling on for the
                                   # compiled executable.  Several default
                                   # profiling setups are provided by this
                                   # script.
  "cc|c=s"      =>  \$main::cc,          # Name of your compiler (overwrites)
  "--cflags=s"  =>  \$addcflags,   # Compiler options
  "--libs=s"    =>  \$addlibs,     # Additional libraries to link
```

```
  "--incls=s"   =>  \$addincls,    # Additional includes
  "--defines=s" =>  \$adddefines,  # Additional defines
  "dmalloc|D"   =>  \$dmalloc,     # User _must_ have dmalloc libraries
                                   # installed! (for memory debugging)
  "nowarn|N"    =>  \$nowarn,      # Turn off setup messages for using
                                   # dmalloc libraries
  "help|h|?"    =>  \$help_flag,
/;

use Getopt::Long;
Getopt::Long::Configure( qw/no_ignore_case_always bundling/ );
my($cmd_line) = GetOptions ( eval($all_options) );

# First order of business is to see if help was requested.  If so, simply
# print out how Getopt::Long was called.  This isn't beautiful, but it
# requires the least maintenence while options are added.
if ($help_flag) {
  print "Outputting Getopt::Long configuration.  \nSee " .
      "http://www.perldoc.com/perl5.6/lib/Getopt/Long.html for " .
      "more information.\n";
  print $all_options;
  print <<HELP_MSG

Notes:
  For options that take an optional string (as indicated by :s after the
  option name), omitting that string makes it empty in the resulting
  Makefile.  Providing the string overwrites the values provided by this
  script.

  For options that take a manditory string, the default behavior is to
  append that string to the existing value.

Makefile not written.
HELP_MSG
;
  exit 1;
}

# Handle profiling if requested
unless ($profile_on) {
  $main::profile = "";
}

if($addcflags) { $main::cflags .= " $addcflags"; }
if($addlibs) { $main::libs .= " $addlibs"; }
if($addincls) { $main::incls .= " $addincls"; }
if($adddefines) { $main::defines .= " $adddefines"; }

# If we have chosen to use the dmalloc library, do all the variable
# mashing here:
if ($dmalloc) {
  $main::libs .= " -ldmalloc";
  $main::defines .= " -DDMALLOC -DDMALLOC_FUNC_CHECK";
  $exeshellargs = "\t\@dmalloc -b -l dmalloc.txt -i 2 high -p " .
    "check-funcs -p check-heap -p log-blocks > ./env_setup\n" .
    "\t\@echo \"rm ./env_setup\" >> ./env_setup\n";
  if (!$nowarn) {
    $exeshellargs .= "\t\@echo\n" .
    "\t\@echo \"Please type . ./env_setup to set your environment\"\n" .
    "\t\@echo \"for memory debugging.  Run the debugger from the\"\n" .
    "\t\@echo \"shell you type this in.\"\n" .
    "\t\@echo\n" .
    "\t\@echo \"Note:  You _must_ have dmalloc properly set up in\"\n" .
    "\t\@echo \"       your shell environment for this library to\"\n" .
    "\t\@echo \"       function properly!\"\n" .
    "\t\@echo\n";
  }
}

# Now, write out the Makefile:
print MAKEFILE <<CONTENTS

DEBUG = $main::debug
PROFILE = $main::profile
BASE = $main::base
CC = $main::cc
CFLAGS = $main::cflags
EXEFILE = $exefile
INCLS = $main::incls
LIBS = $main::libs
VPATH = $vpath
DEFINES = $main::defines

OTHEROBJS = $otherobjs
MAINOBJS = $mainobjs
HEADERS = $headers

.c.o:
  \@\$(CC) -c \$(DEFINES) \$(CFLAGS) \$(INCLS) \$<
  \@echo "Compiling \$*.o"

\$(EXEFILE): \$(OTHEROBJS) \$(MAINOBJS) \$(HEADERS)
  \@echo "linking ..."
  \$(CC) \$(CFLAGS) \$(LIBS) -o \$(EXEFILE) \$(MAINOBJS) \$(OTHEROBJS)
CONTENTS
;
if ($exeshellargs) {
  print MAKEFILE "$exeshellargs\n";
}

print MAKEFILE <<CONTENTS

\$(MAINOBJS): \$(OTHEROBJS) qdb_check.h qdb_check.c
  \$(CC) -c \$(DEFINES) \$(CFLAGS) \$(INCLS) \$*.c

$miscdeps

clean:
  rm -f *.o core
CONTENTS
;

my(@work_list) = split(/:/, $vpath);
for (@work_list) {
  my(@work_list_2) = glob("$_/*.o");
  for (@work_list_2) {
    print MAKEFILE "\trm -f $_\n";
  }
}
print MAKEFILE "\n\n";

print "Done making makefile, cleaning old distribution and exiting.\n";
system("make clean > /dev/null");

exit(0);
```

.qdb_checkrc

```
# actual key value pairs are indicated by having a line beginning with the
# '#' character, and immediately followed by the key.  The _next_ line must
# contain the value of interest, and if there are multiple values, they will
# be read until a line that begins with a '#' is once again encountered.
# (Blank lines will be discarded)
#
# If exactly one value is anticipated, then any line _after_ the values is
# read will be ignored.  Regardless, it will be safest to begin any
# comment line with a '#<space>' combination, since these are guaranteed
# to be ignored.

# The base path of the quantum chemistry database
#db_path
/home/radke/makecd/qdb

# The user name of the program, this could also be retrieved from the
# environment, or set with a configuration utility
#user_name
radke

# When all calculations for a given fragment are complete, the input
# server will notify the following e-mail address, so the force field
# can be finally collated
radke@kilauea.colorado.edu

# The following four will be translated to the local ab initio program's
# input format before submitting the jobs.
#preopt_method
RB3LYP

#preopt_basis
6-31g(d)

#final_method
RB3LYP

#final_basis
6-311+g(2d,p)

#torsion_final_basis
6-31+g(d,p)

#special_flags
nosymm

# Note that this variable indicates not only the name of the local ab
# initio program, but also the command line invocation of it.
#local_ab_initio_program
g98

# Having access to the suffix makes handling the output quite easier, it
# can later be set with a make configure or some other such program
#ab_initio_suffix
com

# The following options relate to the local computing environment, relative
# to the machine that qdb_check is originally run on.  Note that it _may_
# be assumed that the first available_host is some kind of 'master_server'
# that will be used as a clearinghouse for files (i.e. it has access to
# the other host's directories).  The scratch disk on Tumtum is currently
# full, add it back when this is rectified.
#available_hosts
jabberwock
bandersnatch
borogove
beamish
brillig
frabjous
frumious
jubjub
manxome
slithy
tulgey
uffish

#host_connect_method
ssh

# The original program is developed with qsub active, other options may
# be added.
#local_que_method
qsub

# This value is obsolete (I think) but is left here to avoid the possiblity
# of breaking anything that's currently working.
#submission_daemon_path
/home/radke/dev/ff/qdb

# This value is used to determing who is serving the database.
#query_server_host
localhost

# The port listed originally (5561) is read by both the query server, and
```

```
# by any clients that need to hook up to it.  Note that 5561 was an
# unregistered port according to the IANA as of 7-2001
#query_server_port
5561

# This is the range after which asymmetry is assumed to 'inconsequential'
# to properties of atoms and bonds.  While it may be tempting to set it
# to a very large number, don't!  For example, qdb_check uses it to
# determine how far from torsions no asymmetry can exist (without
# demanding an identical match from the database).  If it is set to a
# large value, the program will happily insist that the entire input
# structure must be calculated (along with every torsion within it)
# before it can construct a force field.  This is clearly not the right
# thing to do.  Decent values to choose are:
# 0    Not so good, only the actual atoms on atoms 2 and 3 of a
#    dihedral are checked.
# 1    This should actually be quite good, and would include atoms
#    1, 2, 3, and 4 of a dihedral
# 2    This is anticipated to be nearly perfect, and includes all
#    atoms out to 1 bond on either side of the dihedral
# 3    I would consider this to be a practical maximum
#asymmetry_range
2

# What tolerance is 'acceptable' to indicate a qcode match.  The integer
# portion of the number is 1 + the number of bonds the connectivity and
# electronegativities need to be identical (i.e., 0 wouldn't even require
# the starting atom to be the same).  The decimal portion represents a
# tolerance for the match _past_ the integer range, and can be thought
# of as a 'percentile'.  It also has it's roots in the p function that
# chemists are familiar with (pH, pKa), and as such, values much past
# .4 are very close, and .6 are ridiculously demanding, a default of
# 4.2 - 4.4 is recommended.  For precise details on the determination
# of that number, see get_qcode_deviance() in qdb_shared_functions.c
# (be careful when editing this function, it is used in a perl XSUB
# as well, and changing either the calling convention, or the return
# type will most likely break the XSUB, which is a bit of a bear to
# work with, especially if you're not familiar with the perl API)
#tolerance
4.35

# The following value determines the number of points torsion_driver.pl
# requires before it finishes it's job.  The maximum spacing in the
# calculations will then be 360/<torsion_sampling_rate>
#torsion_sampling_rate
24

# The following value determines the minimum sampling angle (in degrees).
# Some very stiff torsions will make the torsion driver continue to
# bifurcate until it reaches this minimum angle.
#minimum_sampling_angle
2

# The last torsion_driver.pl constant we need is the energy cutoff.
# conformations above this energy will be 'discarded' in the final
# torsion, since they represent 'unphysical' values.  The units on this
# value are kcal/mole, and this value could conceiveably be modified by
# a configuration program, depending on the temperature of the simulation
# the user desires.
#torsion_energy_cutoff
20
```

format_for_g98.pl

```
#! /usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This program (basically a translator) takes exactly one argument, which is
# a .raw format input file, and outputs a .com file (for input) to the
# same directory.  It is designed to be called from qdb_calculate.pl,
# but can be called from anywhere, though it will rely on a .qdb_checkrc
# file being present.  The synopsis is:
# format_for_g98 <working_directory> <input_file_name> <output_base_name>
# Note:  The calling program can discover the suffix from the .qdb_checkrc
# file.

# Now, get the required data from the .qdb_checkrc file
require('../general/rc_file_handling.pl');

open(RCFILE, "<.qdb_checkrc") or die
  "Unable to open .qdb_checkrc in format_for_g98.pl\n";

chdir("$ARGV[0]") or die
  "Unable to set working directory to $ARGV[0] in format_for_g98.pl\n";

open(INFILE, "<$ARGV[1]") or die
  "Unable to open file $ARGV[0]/$ARGV[1] for reading in format_for_g98.pl\n";

open(OUTFILE, ">$ARGV[2].com") or die
  "Unable to open file $ARGV[0]/$ARGV[2].com for writing in " .
  "format_for_g98.pl\n";

$preopt_method = &read_scalar(RCFILE, "preopt_method");
defined($preopt_method) or die
  "Unable to find preopt_method in .qdb_checkrc file ... exiting\n";

$preopt_basis = &read_scalar(RCFILE, "preopt_basis");
defined($preopt_basis) or die
  "Unable to find preopt_basis in .qdb_checkrc file ... exiting\n";

$final_method = &read_scalar(RCFILE, "final_method");
defined($final_method) or die
  "Unable to find final_method in .qdb_checkrc file ... exiting\n";

$final_basis = &read_scalar(RCFILE, "final_basis");
defined($final_basis) or die
  "Unable to find final_basis in .qdb_checkrc file ... exiting\n";

$special_flags = &read_scalar(RCFILE, "special_flags");
defined($special_flags) or die
  "Unable to find special_flags in .qdb_checkrc file ... exiting\n";

close(RCFILE);

@inlist = <INFILE>;
close(INFILE);
for (@inlist) { chomp($_); }

# Ok we now have all we need to finish.
print OUTFILE "%mem=128MB\n";
print OUTFILE "%chk=master.chk\n\n";

print OUTFILE "#p $final_method/$final_basis//" .
        "$preopt_method/$preopt_basis test $special_flags\n\n";

print OUTFILE "No comment\n\n";

print OUTFILE "0 1\n";
print OUTFILE join("\n", @inlist);

print OUTFILE "\n\n--Link1--\n";
print OUTFILE "%mem=128MB\n";
print OUTFILE "%chk=master.chk\n";
print OUTFILE "%nosave\n\n";

print OUTFILE
  "#p $final_method/$final_basis " .
  "test scf=tight pop=chelpg geom=allcheck guess=read nosymm\n\n";

print OUTFILE "0 1\n\n";
```

format_connectivity.sh

```
#! /usr/bin/sh

# This script will take a gaussian *.com file and convert the connectivity
# to a format required by matt's current simulation code.

if [ $# -ne 1 ]; then
  echo "Usage:"
    echo "    "$0" <inputfilename>"
    exit 0
  fi

if [ ! -f $1 ]; then
  echo $1" is not a file ... exiting"
  exit 0
fi

# The first task is to eliminate all lines _not_ concerning connectivity.
awk \
  'BEGIN {
    read_now = 0
  }
  { if ( read_now == 0 && $1 == 1 ) { read_now = 1 }
  }
  { if ( read_now == 1 && $1 != "" ) {
    print
    }
  }
  ' $1 > tmp.$$

# The following awk program formats the connectivity section of a *.com
# file to the generic format needed for the simulation code, and sends it
# to stdout.

awk \
  'BEGIN {
  }
  { if ( NF > 1 ) {
    this_field = 2
    while ( this_field <= NF ) {
      printf("%d %d %s\n", $1, $(this_field), $(this_field + 1))
      this_field += 2
      }
    }
  }
  END {
  }' tmp.$$

rm tmp.$$
```

kqueryserver.sh

```
#! /bin/sh
ssh -n `cat /scratch/radke/qdb/control/qdb_query_server.host` kill -SIGQUIT
`cat /scratch/radke/qdb/control/qdb_query_server.pid`


reghosts.sh
#!/bin/sh

HOSTS="jabberwock bandersnatch beamish borogove brillig frabjous frumious
jubjub manxome slithy tulgey tumtum uffish"

for name in $HOSTS
do
  fullname="${name}.colorado.EDU"
  ssh $fullname
done
```

qdb/qdb_maintenance_utiliti

es

qdb_utilities.pl

```
#!/usr/bin/perl -wT

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

package main;

eval { require 5.6.1 }
  or die <<MESSAGE;
###################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
###################################################################
MESSAGE

use strict;

# Before proceeding, clean up our environment so we can run external
# programs
require '../../general/clean_environment.pl';
full_env_clean();

# The purpose of this script is to execute various actions on the
# quantum database.  It is originally written because there was a need
# to enter my own 'r' or 's' descriptors in the database, but it
# should include modes to verify each entry, correct each entry (for a
# particular _specific_ type of thing, such as repairing connectivity,
# or descriptor information), and offer summaries for the database.
# It may also in the future be able to provide qdb performance
# benchmarks.

# The design is such that it simply wraps a whole slew of other small
# programs or scripts, enabling it to call any of them 'properly',
# leaving the task of understanding how to use them to the program
# itself.  As such, the user should never need to run the programs.
# For the sanity of the user, all of these programs will be kept in
# ./utilities, so the directory looks nice and uncluttered.

# Note: Any commands given in update mode will be executed without
# verification.  If it grows to the point where it's 'too easy' to
# mess up the database in update mode, a verify step, and a -f (force)
# options should be added.

# Special variables used only in dealing with command line options:
my($verify) = 0;
my($update) = 0;
my($summary) = 0;
my($s_descriptor) = 0;
my($frozen_bonds) = 0;
my($charges) = 0;
my($torsions) = 0;
my($all_elements) = 0;
my($qcodes) = 0;

my($help_flag) = 0;

my($all_options) = q/
  "verify|v"    =>  \$verify,       # Verify mode
  "update|u"    =>  \$update,       # Update mode
  "summary|s"   =>  \$summary,      # Summarize mode
  "qcodes|q"    =>  \$qcodes,       # Qcodes
  "descrip|d"   =>  \$s_descriptor, # Stereochemical descriptor
  "freeze|f"  =>  \$frozen_bonds, # Which bonds to freeze
  "charges|c=s" =>  \$charges,      # Which charges to check or update
  "torsions|t"  =>  \$torsions,     # This will work in verify mode
                                   # _only_, and will check the
                 # torsions directory to make
                 # certain it is complete
  "all|a"       =>  \$all_elements, # check all database elements
  "help|h|?"    =>  \$help_flag,
/;

use Getopt::Long;
Getopt::Long::Configure( qw/no_ignore_case_always bundling/ );
my($cmd_line) = GetOptions ( eval($all_options) );

# First order of business is to see if help was requested.  If so, simply
# print out how Getopt::Long was called.  This isn't beautiful, but it
# requires the least maintenance while options are added.
if ($help_flag) {
  print <<HELP_MSG;
Usage:
  $0 [MODE] [ACTION] [OPTIONS]

  Where MODE is [v|u|s] for verify, update, and summarize respectively.

  ACTIONS are the type of action to be performed, see the dump from
  Getopt::long that follows for specific details

  OPTIONS are modifiers to ACTION and as such, may only be relevant if
  certain actions have been specified

  Outputting Getopt::Long configuration.  See
  http://www.perldoc.com/perl5.6/lib/Getopt/Long.html for more
  information.

HELP_MSG

  print $all_options;
  print <<HELP_MSG;

Nothing done
HELP_MSG

  exit 1;
}

if ( $verify + $update + $summary != 1 ) {
  die "Please specify exactly _1_ mode of operation (v, u, or s), " .
  " exiting.\n";
}

# Check if all elements were requested, if so, set all relevant
# variables to 1
if ($all_elements) {
  $s_descriptor = 1;
  $frozen_bonds = 1;
  unless ($charges) {
    $charges = "chelpg";  # Use chelpg as default, since it's the only one we
                          # use at the moment.
  }
  $torsions = 1;
  if ($update) {
  $torsions = 0;
  }
  $qcodes = 1;
}

if ($torsions and $update) {
  die "Torsion directories can only be verified, not updated, " .
  "exiting\n";
}

# The error checking should be done, proceed with initializing the
# directory list, then calling whatever external program is necessary

require "../../general/rc_file_handling.pl" or die
  "Unable to require ../../general/rc_file_handling.pl ... exiting\n";

open ('RCFILE', '../.qdb_checkrc') or die
  "Unable to open ../.qdb_checkrc ... exiting\n";

my($db_path) = &read_scalar('RCFILE', "db_path");
defined($db_path) or die
  "Unable to initialize db_path from ../.qdb_checkrc ... exiting\n";

my($ab_initio_program) = &read_scalar('RCFILE', "local_ab_initio_program");
defined($ab_initio_program) or die
  "Unable to initialize the local ab_initio_program from ../.qdb_checkrc " .
  "... exiting\n";

close('RCFILE');

# Launder our ab_initio_program before requiring it.

($ab_initio_program) = $ab_initio_program =~ m/([\w]+)/;
require "../../perl_modules/${ab_initio_program}_functions.pl" or die
  "Unable to require ../../general/rc_file_handling.pl ... exiting\n";

my($starting_path) = $ENV{'PWD'};

# Launder it ... it is needed to run the utilities
```

```
$starting_path =~ m%[\w/]+%;  # %'s used since / is in the pattern.
$starting_path = $&;

# Launder $db_path, as it is currently tainted.  If a malicious user
# is able to change values in the .qdb_checkrc file, the program will
# either report incorrect results (in verify mode) or it could potentially
# write various files into the target directory.  This possible breach
# will be handled by quitting the program immediately if one of the
# following conditions are true:
# 1)  There is no control directory in the db_path
# 2)  There are directories (besides control) whose names do not match
#     the pattern /((C([2-9]|[\d]*))?(H([2-9]|[\d]*))?(N([2-9]|[\d]*))?(O([2-
9]|[\d]*))?|Other)-[\d]+/
#     I apologize for the 'messiness' of that pattern, but that _should_
#     only match directories whose name has been formatted by
#     qdb_calculate.pl (as determined from it's own function
#     get_CHNO_name()).
# 3)  The directory is empty (besides control)
# I will assume that that is 'enough' security.  The program will then
# change it's own operating directory to that one, so there can be no
# writes outside of the quantum databse.  This should prevent any 'tricky'
# damage to anything but the quantum database.

$db_path =~ m%[\w/]+%;      # %'s used since / is in the pattern.
$db_path = $&;

opendir('DB_DIR', $db_path) or die
  "Cannot open database directory  \"$db_path\" ... exiting\n";

my(@work_list) = readdir('DB_DIR');
closedir(DB_DIR);

# Hash it for faster handling
my(%db_directory);
for (@work_list) {
  $db_directory{$_} = 1;
}

# Remove spurious entries
delete ( $db_directory{"."} );
delete ( $db_directory{".."} );

# Begin security checking
if (exists($db_directory{"control"})) {
  delete($db_directory{"control"});
} else {
  die "No control directory found in db_path ($db_path), this is not " .
    "the correct directory, and may indicate a (lame) attempt to " .
    "make this program do something insecure ... exiting\n";
}

@work_list = keys(%db_directory);

for (@work_list) {
  if ( $_ !~ /^(^(C([2-9]|[\d]*))?(H([2-9]|[\d]*))?(N([2-9]|[\d]*))?(O([2-
9]|[\d]*))?|Other)-[\d]+$/ ) {
  die "A sub_directory of db_path ($db_path), $_ does not match " .
    "an approved name for a quantum database directory, and " .
    "may indicate a (lame) attempt to make this program do " .
      "something insecure ... exiting\n";
  }
}

if ( scalar(@work_list) == 0 ) {
  die "The db_path ($db_path) is empty aside from a control " .
  "directory, I'm not certain this is the directory I want to be " .
    "in.  If this directory is correct, please make initial entries " .
    "into the database ... exiting \n";
}

# Now ... (finally!) we appear to fully secured.  Change to that directory,
# and never look back.  Go on with processing the commands given to us

chdir($db_path) or die
  "Unable to chdir to $db_path ... exiting\n";

# Other command line checking
# First, make certain the charges are one of the types recognized

if ( $charges ) {
  # Launder it, as we need this value to determine which kinds of charges
  # to retrieve later
  my ($tmp) = $charges;
  ($charges) = $charges =~ /([\w]+)/;
  if ( $charges ne $tmp ) {
  die "Insecure or unrecognized charge type $charges.  Exiting\n";
  }

  $charges = lc($charges);
  unless (
      $charges eq "chelpg"
      # or $charges eq "blah" ... other options follow here
     ) {
  print "Unrecognized charge type $charges.  Exiting\n";
  exit;
  }
}

if ($verify) {
  # This section is designed to check all of the directories in the
  # database - to make certain that each directory is complete, and
  # to potentially recommend an action for directories that are
  # missing information.

  print "Verifying quantum database integrity ...\n";

  if ($s_descriptor) {
  print "Checking stereochemical descriptors.\n";
  }

  if ($frozen_bonds) {
  print "Checking frozen bonds.\n";
  }

if ($charges) {
print "Checking $charges charges.\n";
}

if ($torsions) {
print "Checking torsions directories\n";
}

if ($qcodes) {
print "Checking Qcodes files\n";
}

my($this_dir);
my($error_flag) = 0;

for (keys(%db_directory)) {
$this_dir = $_;

if ( -e "$this_dir/is_hosed" ) {
  print "\nDirectory $this_dir contains the file is_hosed.  " .
      "Despite its silly name, this indicates that " .
      "qdb_input_server.pl was interrupted while trying to " .
    "update this directory.  The directory should either " .
    "be removed, or if you're _certain_ it's correct, " .
    "is_hosed can simply be removed manually from this " .
    "directory.\n\n";
  $error_flag++;
}

if ( ! -e "$this_dir/Connectivity.raw") {
  print "\nDirectory $this_dir does not contain the file " .
      "Connectivity.raw.  This file is necessary, and " .
      "this error indicates a corruption of the given " .
    "directory\n\n";
  $error_flag++;
}

if ( ! -e "$this_dir/Original_structure.raw") {
  print "\nDirectory $this_dir does not contain the file " .
      "Original_structure.raw.  This file is necessary, and " .
      "this error indicates a corruption of the given " .
    "directory\n\n";
  $error_flag++;
}

if ( ! -e "$this_dir/Qcodes") {
  print "\nDirectory $this_dir does not contain the file " .
      "Qcodes.  This file is necessary, and " .
      "this error indicates a corruption of the given " .
    "directory\n\n";
  $error_flag++;
}

if ( ! -e "$this_dir/Initial_optimization.com") {
  print "\nDirectory $this_dir does not contain the file " .
      "Initial_optimization.com.  This file is necessary, and " .
      "this error indicates a corruption of the given " .
    "directory.  More specifically, there was never a job " .
    "(to be submitted to an ab intio quantum chemistry " .
    "package, i.e. g98) specified within the directory.\n\n";
  $error_flag++;
}

if ( ! -e "$this_dir/Initial_optimization.log") {
  print "\nDirectory $this_dir does not contain the file " .
      "Initial_optimization.log.  This file is necessary, and " .
      "this error indicates a corruption of the given " .
    "directory.  More specifically, if there was a job " .
    "(corresponding .com file) submitted, it was never " .
    "finished, or the finished calculation was never " .
    "entered into the directory.  It _may_ be possible " .
    "that the job is currently running somewhere, and the " .
    "database is \"waiting\" for the results.  Checking " .
    "this condition, however, is beyond the scope of this " .
    "utility.\n\n";
  $error_flag++;
}

  unless (is_finished("$this_dir/Initial_optimization.log")) {
    print "\nFile $this_dir/Initial_optimization.log does not " .
        "represent a completed calculation.  Make certain that " .
        "the calculation is not presently running.  If it is not, " .
        "please check the partial .log file, and determine the " .
        "reason the job did not finish.  The corresponding .com " .
        "file may need to be modfied to allow the calculation to " .
        "finish, or the host that ran the job may have crashed " .
        "during the calculation.\n\n";
    $error_flag++;
}

# Handle stereochemical descriptors file
if ($s_descriptor) {
  my($command) = "${starting_path}" .
              "/utilities/generate_stereochemical_descriptors " .
              "$this_dir 0";
  open (TMP, "$command |") or
  die "Unable to run generate_sterochemical_descriptors";
  my($work_string) = join("", <TMP>);
  close(TMP);
  if ($?) {
  my($work_string_2);
  my($this_error) = 0;
  open('TMP', "<$this_dir/Stereochemical_descriptors") or
    $this_error = 1;
  unless ($this_error) {
    $work_string_2 = join("", <TMP>);
    close('TMP');
  }

  if ($this_error or $work_string ne $work_string_2) {
    print "Entry $this_dir contains a bad (or no) " .
        "Stereochemical_descriptors file (execute\n\"$0 -ud".
        "\" to repair this problem.\n";
    $error_flag++;
```

```
  }

  }
}

# Handle frozen bonds request
if ($frozen_bonds) {
  my($command) = "${starting_path}" .
               "/utilities/determine_frozen_bonds " .
               "$this_dir";
  open (TMP, "$command |") or
  die "Unable to run generate_sterochemical_descriptors";
  my($work_string) = join("", <TMP>);
  close(TMP);

  if ($?) {
  my($work_string_2);
  my($this_error) = 0;

  open('TMP', "<$this_dir/Frozen_bonds") or
    $this_error = 1;

  unless ($this_error) {
    $work_string_2 = join("", <TMP>);
    close('TMP');
  }

  if ($this_error or $work_string ne $work_string_2) {
    print "Entry $this_dir contains a bad (or no) " .
        "Frozen_bonds file (execute\"$0 -uf".
        "\" to repair this problem.\n";
    $error_flag++;
  }

  } else {
  print "Was unable to determine the proper Frozen_bonds\n" .
        "file for $this_dir.  This is most likely an\n" .
      "installation problem, but may also indicate that\n" .
      "the database is troubled.  Finally, this error may\n" .
      "indicate a logical error in the construction of\n" .
      "this utility\n";
  $error_flag++;
  }
}

# Handle requested charges
if ($charges) {
  my($command) = "${starting_path}" .
               "/utilities/get_${charges}_charges.pl " .
               "$db_path/$this_dir/Initial_optimization.log";

  # See comments in update section on charges for commentary on
  # this design.

  chdir("$starting_path/utilities");
  open (TMP, "$command |") or
  die "Unable to run get_${charges}_charges.pl";
  my($work_string) = join("", <TMP>);
  close(TMP);
  chdir($db_path);

  if ($?) {
  my($work_string_2);
  my($this_error) = 0;

  open('TMP', "<$this_dir/charges.$charges") or
    $this_error = 1;

  unless ($this_error) {
    $work_string_2 = join("", <TMP>);
    close('TMP');
  }

  if ($this_error or $work_string ne $work_string_2) {
    print "Entry $this_dir contains a bad (or no) " .
        "charges.$charges file (execute\n\"$0 -u -c$charges".
        "\" to repair this problem.\n";
    $error_flag++;
  }
  } else {
  print "Was unable to determine the proper charges.$charges\n" .
        "file for $this_dir.  This is most likely an\n" .
      "installation problem, but may also indicate that\n" .
      "the database is troubled.  Finally, this error may\n" .
      "indicate a logical error in the construction of\n" .
      "this utility\n";
  $error_flag++;
  }
}

# Handle torsions directories
if ($torsions) {
  my($command) = "${starting_path}" .
               "/utilities/check_torsions.pl " .
               "$db_path/$this_dir 1";

  # See comments in update section on charges for commentary on
  # this design.

  chdir("$starting_path/utilities");
  open (TMP, "$command |") or
  die "Unable to run check_torsions.pl";
  my(@output) = <TMP>;
  close(TMP);
  chdir($db_path);
  if ($?) {
  # Process output of check_torsions here
  foreach (@output) {
    print $_;
    print "\n";
    $error_flag++;
  }

  } else {
    print "$command returned with unknown error status\n";
    $error_flag++;
    }
  }

  # Handle qcodes
  if ($qcodes) {
    my($command) = "${starting_path}" .
                 "/utilities/generate_qcodes $this_dir";
    open (TMP, "$command |") or
    die "Unable to run generate_qcodes";
    my($work_string) = join("", <TMP>);
    close(TMP);
    if ($?) {
    my($work_string_2);
    my($this_error) = 0;
    open('TMP', "<$this_dir/Qcodes") or
      $this_error = 2;
    unless ($this_error) {
      $work_string_2 = join("", <TMP>);
      close('TMP');
    }

    if ($this_error == 2) {
      print "There was no file $this_dir/Qcodes found, which " .
          "indicates that is was likely never created in $this_dir.  ".
        "This is a critical problem, please execute \"$0 -uq\" " .
        "to repair this problem)\n";
      $error_flag++;
    }

    if ($work_string ne $work_string_2) {
      print "Entry $this_dir contains a bad Qcodes file " .
          "(execute\"$0 -uq.\" to repair this problem).  Note " .
        "that the current program only checks to see if the " .
        "Qcodes file exactly matches what this machine calculates " .
        "them to be.  This is done due to a current limitation in " .
        "the charge query section of qdb_query_server, but should " .
        "be repaired in a later version.\n";
      $error_flag++;
    }

    }
  }

  # End optional argument handling

  }
  # Finally, output any limitations to the verify portion of this
  # utility --- this section can be useful to guide future
  # development.  Please add any limitations discovered in the
  # future to the following message.

  if ($error_flag) {
  print "Database verification completed.  There were " .
        "$error_flag errors detected.  See above messages for " .
      "specific instructions and/or advice\n";
  if ( $error_flag > 10 ) {
    print "Your database is hosed and needs serious help! :^)";
  }
  } else {
  print "Database verification completed with 0 errors!\n";
  }

  print <<LIMITATIONS;

Note:  The verify utility has specific limitations.  Unless an extra
option was specified, it does not check the following:

1)  It does not check to verify the Connectivity or Geometry of the
    molecule in the specified directory is "reasonable".
2)  It does not check that the results of the quantum chemical calculations
    are "reasonable".
3)  It does not check to verify that there is a "proper" number of
    qcodes for each molecule.
4)  It does not check the permissions of the files, except in so far as it
    will fail in it's workings if it doesn't have permission to check
    for the existence of the files.

Other limitations:
LIMITATIONS

  unless ($s_descriptor) {
  print <<LIMITATIONS;
    It does not check (unless requested, use option -d to get this
    information) that any stereochemical descriptors that should be in
    the current directory are actually there.

LIMITATIONS
  }

  if ($charges) {
  print <<LIMITATIONS;
    This program will only check for the type of charges specified.
    Likewise, it will only update the type of charges specified.

LIMITATIONS
  }

  if ($qcodes) {
  print <<LIMITATIONS;
    This program only checks to see if the existing Qcodes file is an
    exact match of what this machine calculates the qcodes should be.
    This behaviour is due to a current limitation in the charges matching
    section of qdb_query_server.pl, and should be repaired in a later
    version.

LIMITATIONS
  }
  # Only one mode of execution can be active at a time, so send
  # a successful exit to the calling program;
  exit (0);

} elsif ($summary) {
```

```
# This section generates some statistics on the quantum chemistry
# database, and outputs them.  It is (like many other areas of
# the package) likely to grow as database management gets more complex.

@work_list = keys(%db_directory);
my($this_dir);
print "Scanning " . scalar(@work_list) . " database fragments, this ".
"may take a bit, please wait ...\n";

# Variables to be tracked through the scanning
my($total_MB);
my($total_GB);
my($atom_count);
my($bond_count);
my($min_qcode_depth) = 1000000;
my($max_qcode_depth) = 0;
my($file_count) = 0;
my($torsion_dir_count) = 0;
my($torsion_single_point_count) = 0;

for ( keys(%db_directory) ) {
$this_dir = $_;

open ('TMP', "<$this_dir/Connectivity.raw") or die
  "Unable to open $this_dir/Connectivity.raw for " .
  "reading, please run $0 -v to get more information " .
  "on this error\n";

# Note:  I'm not certain why this is the case, but the stat
# call _fails_ if it's called as "stat('TMP')".  My understanding
# is that TMP is a bareword, and to be avoided, but I'll do it
# however it works for now :-).

@work_list = stat(TMP) or die
  "Unable to get statistics for $this_dir/Connectivity.raw, " .
  "it's not certain what this error means.  Specifically, we " .
  "were able to open the file for reading, but _unable_ to " .
  "get statistics on it\n";
$total_MB += $work_list[7] / 1024 / 1024;

while ( <TMP> ) {
  $bond_count++;
}
close('TMP');

open ('TMP', "<$this_dir/Original_structure.raw") or die
  "Unable to open $this_dir/Original_structure.raw for " .
  "reading, please run $0 -v to get more information " .
  "on this error.\n";

@work_list = stat(TMP) or die
  "Unable to get statistics for " .
  "$this_dir/Original_structure.raw, " .
  "it's not certain what this error means.  Specifically, we " .
  "were able to open the file for reading, but _unable_ to " .
  "get statistics on it.\n";
$total_MB += $work_list[7] / 1024 / 1024;

while ( <TMP> ) {
  $atom_count++;
}
close('TMP');

open ('TMP', "<$this_dir/Qcodes") or die
  "Unable to open $this_dir/Qcodes for " .
  "reading, please run $0 -v to get more information " .
  "on this error.\n";

@work_list = stat(TMP) or die
  "Unable to get statistics for $this_dir/Qcodes.raw, " .
  "it's not certain what this error means.  Specifically, we " .
  "were able to open the file for reading, but _unable_ to " .
  "get statistics on it.\n";
$total_MB += $work_list[7] / 1024 / 1024;

while ( <TMP> ) {
  @work_list = split;

  if (scalar(@work_list) < $min_qcode_depth) {
  $min_qcode_depth = scalar(@work_list);
  }

  if (scalar(@work_list) > $max_qcode_depth) {
  $max_qcode_depth = scalar(@work_list);
  }
}
close('TMP');

# Now, scan the size of all of the files that we're not getting
# other specific information from;

open ('TMP', "<$this_dir/Initial_optimization.com") or die
  "Unable to open $this_dir/Initial_optimization.com for " .
  "reading, please run $0 -v to get more information " .
  "on this error.\n";

@work_list = stat(TMP) or die
  "Unable to get statistics for " .
  "$this_dir/Initial_optimization.com, " .
  "it's not certain what this error means.  Specifically, we " .
  "were able to open the file for reading, but _unable_ to " .
  "get statistics on it.\n";
$total_MB += $work_list[7] / 1024 / 1024;
close(TMP);

open ('TMP', "<$this_dir/Initial_optimization.log") or die
  "Unable to open $this_dir/Initial_optimization.com for " .
  "reading, please run $0 -v to get more information " .
  "on this error.\n";

@work_list = stat(TMP) or die
  "Unable to get statistics for " .
  "$this_dir/Initial_optimization.log, " .
  "it's not certain what this error means.  Specifically, we " .
  "were able to open the file for reading, but _unable_ to " .
  "get statistics on it.\n";
$total_MB += $work_list[7] / 1024 / 1024;
close(TMP);

$file_count += 5;

# If there's a descriptors file, count the size of that as well.
if ( -e "$this_dir/Stereochemical_descriptors" ) {
  $file_count++;

  open ('TMP', "<$this_dir/Stereochemical_descriptors") or die
  "Unable to open $this_dir/Stereochemical_descriptors for " .
  "reading, please run $0 -vd to get more information " .
  "on this error.\n";

  @work_list = stat(TMP) or die
  "Unable to get statistics for " .
  "$this_dir/Stereochemical_descriptors, " .
  "it's not certain what this error means.  Specifically, we " .
  "were able to open the file for reading, but _unable_ to " .
  "get statistics on it.\n";
  $total_MB += $work_list[7] / 1024 / 1024;
  close(TMP);
}

# If there's a frozen bonds file, count the size of that as well.
if ( -e "$this_dir/Frozen_bonds" ) {
  $file_count++;

  open ('TMP', "<$this_dir/Frozen_bonds") or die
  "Unable to open $this_dir/Frozen_bonds for " .
  "reading, please run $0 -vd to get more information " .
  "on this error.\n";

  @work_list = stat(TMP) or die
  "Unable to get statistics for " .
  "$this_dir/Frozen_bonds, " .
  "it's not certain what this error means.  Specifically, we " .
  "were able to open the file for reading, but _unable_ to " .
  "get statistics on it.\n";
  $total_MB += $work_list[7] / 1024 / 1024;
  close(TMP);
}

# If there are charges files, only register the known types.
if ( -e "$this_dir/charges.chelpg" ) {
  $file_count++;

  open ('TMP', "<$this_dir/charges.chelpg") or die
  "Unable to open $this_dir/charges.chelpg for " .
  "reading, please run $0 -vd to get more information " .
    "on this error.\n";

  @work_list = stat(TMP) or die
  "Unable to get statistics for " .
  "$this_dir/charges.chelpg, " .
  "it's not certain what this error means.  Specifically, we " .
  "were able to open the file for reading, but _unable_ to " .
  "get statistics on it.\n";
  $total_MB += $work_list[7] / 1024 / 1024;
  close(TMP);
}

# Call check_torsions.pl to return any necessary information
my($command) = "${starting_path}" .
               "/utilities/check_torsions.pl " .
             "$db_path/$this_dir 2";

chdir("$starting_path/utilities");
open (TMP, "$command |") or
  die "Unable to run check_torsions.pl";
$torsion_dir_count += <TMP>;
$file_count += <TMP>;
$total_MB += <TMP>;
$torsion_single_point_count += <TMP>;

close(TMP);
chdir($db_path);
}

# Finally, output all of the gathered information
$total_GB = $total_MB / 1024;
$total_GB = int($total_GB * 100 + 0.5) / 100;
$total_MB = int($total_MB + 0.5);
print <<SUMMARY;
Database scanning completed.

Statistics:
The database has a total of $file_count files.
The (primary) database contains $atom_count atoms.
The (primary) database contains $bond_count bonds.
SUMMARY

  if ($min_qcode_depth == $max_qcode_depth) {
  print "The qcode depth is $min_qcode_depth, and is the same for " .
       "each entry in the database\n";
  } else {
  print "WARNING:  The qcode depth is not consistent throughout the " .
       "database, there\n\tmay be subtle errors using this database.\n";
  }

  print <<SUMMARY;
The database has a total of $torsion_dir_count torsions.
The database has a total of $torsion_single_point_count individual torsion
geometries.
The database has a total of $total_MB Megabytes ($total_GB GB) of data.

SUMMARY

} elsif ($update) {
  # This section does various maintenance tasks on the database.  For
  # every specific case handled here, it's likely that there will be
  # a corresponding section in the verify portion of this program,
```

```
# which may be easier to develop first --- then port it's usage to
# this section.
if ($s_descriptor) {
my($this_dir);
@work_list = keys(%db_directory);
print "Repairing stereochemical descriptors for " .
  scalar(@work_list) . " entries, this may take awhile.\n";

# Let the iterations beging

for (keys(%db_directory)) {
  $this_dir = $_;

  my($command) = "${starting_path}" .
             "/utilities/generate_stereochemical_descriptors " .
             "$this_dir 0";
  open (TMP, "$command |") or
  die "Unable to run generate_sterochemical_descriptors";
  my($work_string) = join("", <TMP>);
  close(TMP);
  if ($?) {  # In otherwords, the called program did something

  open('TMP', ">$this_dir/Stereochemical_descriptors") or die
    "Unable to open $this_dir/Sterechemical_descriptors" .
    " for writing ... exiting\n";
  print TMP $work_string;
  close('TMP');
  }
}
}

if ($frozen_bonds) {
my($this_dir);
@work_list = keys(%db_directory);
print "Repairing frozen bonds files for " .
  scalar(@work_list) . " entries, this may take awhile.\n";

# Let the iterations begin

for (keys(%db_directory)) {
  $this_dir = $_;

  my($command) = "${starting_path}" .
  "/utilities/determine_frozen_bonds " .
  "$this_dir";
  open (TMP, "$command |") or
  die "Unable to run determine_frozen_bonds";
  my($work_string) = join("", <TMP>);
  close(TMP);
  if ($?) {  # In otherwords, the called program did something

  open('TMP', ">$this_dir/Frozen_bonds") or die
    "Unable to open $this_dir/Frozen_bonds" .
    " for writing ... exiting\n";
  print TMP $work_string;
  close('TMP');
  }
}
}

if ($charges) {
my($this_dir);
@work_list = keys(%db_directory);
print "Repairing charges ($charges) files for " .
  scalar(@work_list) . " entries, this may take awhile.\n";

# Let the iterations begin

for (keys(%db_directory)) {
  $this_dir = $_;

  my($command) = "${starting_path}" .
  "/utilities/get_${charges}_charges.pl " .
  "$db_path/$this_dir/Initial_optimization.log";

   # Once again, my (brain dead) design requires that the utility
   # be run from it's own directory.  We'll change there temporarily.
   # Eventually, we can correct this problem by adding (globally)
   # to perl's @INC list the directories in the distribution that
   # contain library files.

  chdir("$starting_path/utilities");
  open (TMP, "$command |") or
  die "Unable to run get_${charges}_charges.pl";
  my($work_string) = join("", <TMP>);
  close(TMP);
  chdir($db_path);

  if ($?) {  # In otherwords, the called program did something

  open('TMP', ">$this_dir/charges.$charges") or die
    "Unable to open $this_dir/charges.$charges" .
    " for writing ... exiting\n";
  print TMP $work_string;
  close('TMP');
  }
}
}

if ($qcodes) {
my($this_dir);
@work_list = keys(%db_directory);
print "Repairing Qcodes for " . scalar(@work_list) .
  " entries, this may take awhile.\n";

# Let the iterations beging

for (keys(%db_directory)) {
  $this_dir = $_;

  my($command) = "${starting_path}" .
             "/utilities/generate_qcodes $this_dir";
  open (TMP, "$command |") or
  die "Unable to run generate_qcodes";
    my($work_string) = join("", <TMP>);
    close(TMP);
    if ($?) {  # In otherwords, the called program did something

    open('TMP', ">$this_dir/Qcodes") or die
      "Unable to open $this_dir/Qcodes for writing ... exiting\n";
    print TMP $work_string;
    close('TMP');
    }
  }
  }

  # As a marker, this is where we look for any other relevant repairs
  # that are defined.

} else {
  die "Unknown operating mode specified.  This indicates a program " .
    "bug (unknown condition) ... exiting\n";
}

# The module ../perl_modules/g98_functions uses a my_warn() function.  We'll
# provide our own for this utility.
sub my_warn($) {

  my($message) = shift;

  print "Warning!  $message\n";

}
```

utilities/configure.pl

```
#!/usr/bin/perl -wT

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

package main;

eval { require 5.6.1 }
  or die <<MESSAGE;
#################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
#################################################################
MESSAGE

use strict;

# Before proceeding, clean up our environment so we can run external
# programs
require '../../../general/clean_environment.pl';
full_env_clean();

# This script will configure the makefile for whatever operating system
# we are running on.  It gets this information from the environment
# variable $OSTYPE, which will be laundered and used immediately as
# if it was secure.  I cannot perceive of any circumstances that would
# make this a security problem.

my($os)=$^O;

if ($os =~ /^([-\@\w.]+)$/) {
  $os = $1;  #$os is laundered
} else {
  die "Bad system \"$os\" retrived from environment, exiting";
}

my($path_to_root) = '../../../';

open ("MAKEFILE", ">./Makefile") or die
  "Unable to open makefile for writing, exiting";

print MAKEFILE <<MY_MESSAGE
###################################################################
# This makefile was created by configure.pl.  Any changes to this
# makefile will be overwritten!  Please edit the configure.pl script
# instead.
#
#
MY_MESSAGE
;

# Values for files in this package, these are the mostly likely to
# change regularly, so they're put first
```

```
# In this (specific) implementation, all of the files will be included
# explicitly.
my($vpath) = "../..:../../../general:../../../log2str";
my($mainobjs) = "";
my($headers) = "";
my($otherobjs) = "";
my($exefile) = "";
my($miscdeps)=<<MORESTUFF
# Miscelleneous dependencies here
# Provide explicit pathnames to various object files
../../../general/atom_handling.o:
../../../general/vector_handling.o:
../../qdb_shared_functions.o:
../../../log2str/get_bond_order.o:


MORESTUFF
;

# Default makefile values

# Note:  Just a reminder.  my($variable) variables are local only to this
# scope, which is main.  If another file is 'require' 'ed, it will _not_
# have access to these variables unless they're fully qualified (since
# I'm using strict vars).  Any variables that are changed in the os
# specific instructions must be scoped to main here.

$main::debug = "-g";
$main::profile = "";
$main::base = "";
$main::cc = "cc";
$main::cflags = '$(DEBUG) $(PROFILE)';
$main::libs = "-lm";
$main::incls = "";
$main::defines = "";
my($dmalloc) = "";
my($exeshellargs) = "";
my($nowarn) = "";

# Include any information for system specific options
my ($sys_config_file) = $path_to_root . 'general/os_specific/' .
  $os . '_make.pl';

if ( -r $sys_config_file ) {
  # Announce our finding.  Note, the file itself should provide the
  # newline, and any other additional information to print.
  print "Preparing makefile for $os.  ";

  # And use it
  require $sys_config_file;
} else {
  print "Don't know how to make makefile for $os (i.e., file\n" .
    "\"general/os_specific/${os}_make.pl\" not found).  Using \n" .
    "generic configuration file\n";
  require $path_to_root . 'general/os_specific/generic_make.pl';
}


# I haven't worked out anything too fancy for command line handling, but
# for now, I'll add the options in hodgepodge, and document as I go.  Note
# That command line processing doesn't happen until _after_ the defaults
# are entered.  This allows any of the defaults to be overwritten.  If any
# arguments that normally take a string are left empty, they will also
# be empty in the resulting Makefile.

# Special variables used only in dealing with command line options:
my($addcflags);
my($addlibs);
my($addincls);
my($adddefines);
my($help_flag);
my($profile_on);

my($all_options) = q/
  "debug|d:s"   =>  \$main::debug,
  "profile|p:s" =>  \$main::profile, # Ignored unless -P is also provided
  "P"           =>  \$profile_on,    # Set this to turn profiling on for the
                                     # compiled executable.  Several default
                                     # profiling setups are provided by this
                                     # script.
  "cc|c=s"      =>  \$main::cc,      # Name of your compiler (overwrites)
  "--cflags=s"  =>  \$addcflags,     # Compiler options
  "--libs=s"    =>  \$addlibs,       # Additional libraries to link
  "--incls=s"   =>  \$addincls,      # Additional includes
  "--defines=s" =>  \$adddefines,    # Additional defines
  "dmalloc|D"   =>  \$dmalloc,       # User _must_ have dmalloc libraries
                                     # installed! (for memory debugging)
  "nowarn|N"    =>  \$nowarn,        # Turn off setup messages for using
                                     # dmalloc libraries
  "help|h|?"    =>  \$help_flag,
/;

use Getopt::Long;
Getopt::Long::Configure( qw/no_ignore_case_always bundling/ );
my($cmd_line) = GetOptions ( eval($all_options) );

# First order of business is to see if help was requested.  If so, simply
# print out how Getopt::Long was called.  This isn't beautiful, but it
# requires the least maintenence while options are added.
if ($help_flag) {
  print "Outputting Getopt::Long configuration.  \nSee " .
      "http://www.perldoc.com/perl5.6/lib/Getopt/Long.html for " .
      "more information.\n";
  print $all_options;
  print <<HELP_MSG

Notes:
  For options that take an optional string (as indicated by :s after the
  option name), omitting that string makes it empty in the resulting
  Makefile.  Providing the string overwrites the values provided by this
  script.

  For options that take a manditory string, the default behavior is to
  append that string to the existing value.

Makefile not written.
HELP_MSG
;
  exit 1;
}

# Handle profiling if requested
unless ($profile_on) {
  $main::profile = "";
}

if($addcflags) { $main::cflags .= " $addcflags"; }
if($addlibs) { $main::libs .= " $addlibs"; }
if($addincls) { $main::incls .= " $addincls"; }
if($adddefines) { $main::defines .= " $adddefines"; }

# If we have chosen to use the dmalloc library, do all the variable
# mashing here:
if ($dmalloc) {
  $main::libs .= " -ldmalloc";
  $main::defines .= " -DDMALLOC -DDMALLOC_FUNC_CHECK";
  $exeshellargs = "\t\@dmalloc -b -l dmalloc.txt -i 2 high -p check-funcs " .
    "-p check-heap -p log-blocks > ./env_setup\n" .
    "\t\@echo \"rm ./env_setup\" >> ./env_setup\n";
  if (!$nowarn) {
    $exeshellargs .= "\t\@echo\n" .
    "\t\@echo \"Please type . ./env_setup to set your environment\"\n" .
    "\t\@echo \"for memory debugging.  Run the debugger from the\"\n" .
    "\t\@echo \"shell you type this in.\"\n" .
    "\t\@echo\n" .
    "\t\@echo \"Note:  You _must_ have dmalloc properly set up in\"\n" .
    "\t\@echo \"       your shell environment for this library to\"\n" .
    "\t\@echo \"       function properly!\"\n" .
    "\t\@echo\n";
  }
}

# Now, write out the Makefile:
print MAKEFILE <<CONTENTS

DEBUG = $main::debug
PROFILE = $main::profile
BASE = $main::base
CC = $main::cc
CFLAGS = $main::cflags
EXEFILE = $exefile
INCLS = $main::incls
LIBS = $main::libs
VPATH = $vpath
DEFINES = $main::defines

OTHEROBJS = $otherobjs
MAINOBJS = $mainobjs
HEADERS = $headers

.c.o:
  \@\$(CC) -c \$(DEFINES) \$(CFLAGS) \$(INCLS) \$<
  \@echo "Compiling \$*.o"

all: generate_stereochemical_descriptors determine_frozen_bonds \\
     generate_qcodes

generate_stereochemical_descriptors: generate_stereochemical_descriptors.o\\
  atom_handling.o vector_handling.o get_bond_order.o\\
  qdb_shared_functions.o assign_qcodes.o
  \@echo "linking ..."
  \$(CC) \$(CFLAGS) \$(LIBS) -o generate_stereochemical_descriptors\\
    generate_stereochemical_descriptors.o atom_handling.o\\
    vector_handling.o get_bond_order.o qdb_shared_functions.o\\
    assign_qcodes.o

determine_frozen_bonds: determine_frozen_bonds.o\\
  atom_handling.o vector_handling.o get_bond_order.o\\
  qdb_shared_functions.o assign_qcodes.o
  \@echo "linking ..."
  \$(CC) \$(CFLAGS) \$(LIBS) -o determine_frozen_bonds\\
    determine_frozen_bonds.o atom_handling.o\\
    vector_handling.o get_bond_order.o qdb_shared_functions.o\\
    assign_qcodes.o

generate_qcodes: generate_qcodes.o\\
  atom_handling.o vector_handling.o get_bond_order.o\\
  qdb_shared_functions.o assign_qcodes.o
  \@echo "linking ..."
  \$(CC) \$(CFLAGS) \$(LIBS) -o generate_qcodes\\
    generate_qcodes.o atom_handling.o\\
    vector_handling.o get_bond_order.o qdb_shared_functions.o\\
    assign_qcodes.o



CONTENTS
;
if ($exeshellargs) {
  print MAKEFILE "$exeshellargs\n";
}

print MAKEFILE <<CONTENTS

$miscdeps

clean:
  rm -f *.o core
CONTENTS
;

my(@work_list) = split(/:/, $vpath);
for (@work_list) {
  my(@work_list_2) = glob("$_/*.o");
  for (@work_list_2) {
    print MAKEFILE "\trm -f $_\n";
  }
}
```

```
print MAKEFILE "\n\n";

print "Done making makefile, cleaning old distribution and exiting.\n";
system ("make clean > /dev/null");

exit(0);
```

utilities/determine_frozen_bonds.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* This is a utility to output which bonds should be frozen for any */
/* dihedral scans done in the quantum chemistry database.  */

/* Includes */
#include <stdio.h>
#include <string.h>

#include "../../../general/atom.h"

/* Macro and type definitions */
#ifndef MAXSTR
#define MAXSTR 256
#endif

#ifndef BOOLEAN
#define BOOLEAN
typedef enum {false = 0, no = 0, true = 1, yes = 1} boolean;
#endif

/* Forward prototypes (for functions used from other files).  Note that */
/* we could also include those files headers, but this would require we */
/* fuss with the Makefile (configure.pl) as well.  It seems most sensible */
/* in a monolithic program like this to just declare what we need */
int assign_qcodes(atom *some_atom);

/* And finally, functions that live here */
boolean is_methyl_group(atom *some_atom);

int main (int argc,char *argv[]) {

  /* Variable declarations */
  char db_path[MAXSTR], work_string[MAXSTR], another_string[MAXSTR];
  FILE* work_file;
  boolean do_loop, did_something = no, stay_here, one_aromatic;
  double x, y, z;
  atom *molecule_base = NULL, *work_atom;
  int molecule_size = 0, i, j, offset;
  float bond_order;

  /* Initialization of command line variables */
  if ( argc != 2 ) {
  printf(
       /* Begin Usage message */
       "Usage:\n"
       "\n"
       "%s <database_directory>\n"
       "\n"
       "Where <database_directory> is a fully qualified path to the \n"
       "parent database directory, and:\n"
       "\n"
       "\n", argv[0]);
  error_exit("In short, don't use thie program manually.  It is meant to "
         "be called by qdb_maintenance_utilities.pl");

  }

  if (strlen(argv[1]) > MAXSTR - 1) {
  printf("First argument is too long (> %i) characters", MAXSTR);
  error_exit("");
  }
  strcpy(db_path, argv[1]);

  /* Initialization of command line is finished */

  /* And ... begin our work */
  if ( strlen(db_path) + strlen("Original_structure.raw") + 2 > MAXSTR ) {
  error_exit("Potential string overflow in main()");
  }

  if (! strcpy(work_string, db_path) ) {
  error_exit("Unable to copy string in main()");
  }

  strcat(work_string, "/Original_structure.raw");

  if ( ! (work_file = fopen(work_string, "r")) ) {
  error_exit("Unable to open Original_structure.raw in main()");
  }

  /* Work file is open, start the initializations */
  do_loop = true;

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  while (do_loop) {
  if ( ( fgets(work_string, MAXSTR, work_file) ) == NULL ) { break; }
  if (sscanf(work_string, "%[^,],%lf,%lf,%lf\n",
        another_string, &x, &y, &z) == 4) {
    /* We're reading coordinates, keep going */
    molecule_base = atom_realloc(molecule_base, ++molecule_size,
                strlen(another_string) + 1);

    strcpy(molecule_base[molecule_size - 1].label, another_string);
    molecule_base[molecule_size - 1].coordinates[0] = x;
    molecule_base[molecule_size - 1].coordinates[1] = y;
    molecule_base[molecule_size - 1].coordinates[2] = z;

  } else {
    do_loop = false;
  }

  }

  /* Now ... initialize the atomic numbers and last/next links */
  for (i = 0; i < molecule_size; i++) {
  if ( (molecule_base[i].atomic_number =
      atom_lab_to_num(molecule_base[i].label) ) == 0 ) {
    warn_out("Unknown atom label encountered in main, it's most "
        "likely that the structure file is corrupted");
  }

  if ( i != 0 ) {
    molecule_base[i].previous = molecule_base + i - 1;
  }

  if ( i != molecule_size  - 1 ) {
    molecule_base[i].next = molecule_base + i + 1;
  }
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  /* We're done with the Original_structure.raw file, close it and work */
  /* on the Connectivity.raw file */

  if ( fclose( work_file ) == EOF ) {
  warn_out("Unable to close Original_structure.raw in main(), this "
      "should not be fatal, but is definately a problem");
  }

  if ( strlen(db_path) + strlen("Connectivity.raw") + 2 > MAXSTR ) {
  error_exit("Potential string overflow in main()");
  }

  if (! strcpy(work_string, db_path) ) {
  error_exit("Unable to copy string in main()");
  }

  strcat(work_string, "/Connectivity.raw");

  if ( ! (work_file = fopen(work_string, "r")) ) {
  error_exit("Unable to open Connectivity.raw in main()");
  }

  /* Work file is open, start the initializations */
  do_loop = true;

  while ( do_loop ) {

  if ( (fgets(work_string, MAXSTR, work_file) ) == NULL) { break; }

  /* Read connectivity information */
  if (sscanf(work_string, "%d %d %f\n", &i, &j, &bond_order) == 3 ) {
    atom_connect(molecule_base + i, molecule_base + j, bond_order);
  } else {
    do_loop = false;
  }
  }

  /* And ... close the file */
  fclose(work_file);

  /* We do not need to assign qcodes, since we-re only worrried about the */
  /* connectivity and bond orders */
  if ( assign_qcodes(molecule_base) == 0 ) {
    error_exit("Failed to initialize all qcodes in main()");
  }

  /* At this point, we simply assume the information in the database is */
  /* both complete and correct.  We could do a lot of other checking on */
  /* thinks like making sure the valences are full, making sure the */
  /* formal charges are correctly assigned, and so forth.  The job of */
  /* verifying the integrity of the database information falls on */
  /* ../qdb_utilities, not here, though another program with a similar */
  /* structure may be written to verify the actual structures in the */
  /* database */

  /* Phew!  After all of that setup (which should probably eventually */
  /* be moved to a single function), we're ready to enumerate all of */
  /* the bonds that should be frozen.  Our current (single) criterium */
  /* is that if it's a single bond, and not a methyl group or it should */
  /* be frozen.  Other rules can be added as they come up */

  /* Okies, there is no single criterium anymore.  I've also decided */
```

```
  /* that bonds connecting two phenyl rings should not be frozen.  */

  for (work_atom = molecule_base; work_atom; work_atom = work_atom->next) {

  offset = work_atom - molecule_base;

  stay_here = true;
  one_aromatic = false;
  if (work_atom->atomic_number == 1) {stay_here = false;}
  if (is_methyl_group(work_atom)) {stay_here = false;}
  if (is_aromatic(work_atom)) { one_aromatic = true; }

  if( stay_here ) {
    for (i = 0; i < work_atom->valence; i++) {
    if (work_atom->bond[i]->atomic_number == 1) { continue; }
    if (is_methyl_group(work_atom->bond[i]) ) { continue; }
    if ( one_aromatic && is_aromatic(work_atom->bond[i]) ) {
      continue;
    }

    /* And ... so we only get one of each qualifying bond */
    if ( offset > get_atom_offset(work_atom->bond[i]) ) {
      continue;
    }

    /* The final test */
    if ( work_atom->bond_order[i] != 1 ) { continue; }

    /* If we passed all of the tests, we can print out these */
    /* two atoms */
    printf("%d %d\n", offset,
         get_atom_offset(work_atom->bond[i]) );
    did_something = yes;
    }
  }
  }

  free_molecule(molecule_base);

  if (did_something) { return 1; }

  return 0;
}
```

utilities/generate_qcodes.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* This is a utility to generate what _should_ be the qcodes file in
   the chosen directory.  It is meant to be called by ../qdb_utilities
   for the purpose of verification or repair */

/* According to my dmalloc audit of this program, it is completely */
/* leak free */

/*

generate_qcodes <database_directory>

This program will generate the output text for a qcodes file.
qdb_utilities.pl will call this program to verify or add this file to
the appropriate directory.  <database_directory> is a full path to the
database entry to be read and ouput prepared for.  The program will
use the functions in ../../../general/atom_handling.c for
determination of output.

Other notes:
The program is picky about the pathname it gets, it will do unknown
things if the path has a trailing "/"
*/

/* Includes */
#include <stdio.h>
#include <string.h>

#include "../../../general/atom.h"

/* Macro and type definitions */
#ifndef MAXSTR
#define MAXSTR 256
#endif

#ifndef BOOLEAN
#define BOOLEAN
typedef enum {false = 0, no = 0, true = 1, yes = 1} boolean;
#endif

/* Forward prototypes (for functions used from other files).  Note that */
/* we could also include those files headers, but this would require we */
/* fuss with the Makefile (configure.pl) as well.  It seems most sensible */
/* in a monolithic program like this to just declare what we need */
int assign_qcodes(atom *some_atom);

int main (int argc,char *argv[]) {

  /* Variable declarations */
  char db_path[MAXSTR], work_string[MAXSTR], another_string[MAXSTR];
  FILE* work_file;
  boolean do_loop, did_something = no;
  double x, y, z;
  atom *molecule_base = NULL, *work_atom;
  int molecule_size = 0, i, j;
  float bond_order;

  /* Initialization of command line variables */
  if ( argc != 2 ) {
  printf(
       /* Begin Usage message */
       "Usage:\n"
       "\n"
       "%s <database_directory>\n"
       "\n"
       "Where <database_directory> is a fully qualified path to the parent\n"
       "database directory.\n"
       "\n"
       "\n", argv[0]);
  error_exit("");

  }

  if (strlen(argv[1]) > MAXSTR - 1) {
  printf("First argument is too long (> %i) characters", MAXSTR);
  error_exit("");
  }
  strcpy(db_path, argv[1]);

  /* Initialization of command line is finished */

  /* And ... begin our work */
  if ( strlen(db_path) + strlen("Original_structure.raw") + 2 > MAXSTR ) {
  error_exit("Potential string overflow in main()");
  }

  if (! strcpy(work_string, db_path) ) {
  error_exit("Unable to copy string in main()");
  }

  strcat(work_string, "/Original_structure.raw");

  if ( ! (work_file = fopen(work_string, "r")) ) {
  error_exit("Unable to open Original_structure.raw in main()");
  }

  /* Work file is open, start the initializations */
  do_loop = true;

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  while (do_loop) {
  if ( ( fgets(work_string, MAXSTR, work_file) ) == NULL ) { break; }
  if (sscanf(work_string, "%[^,],%lf,%lf,%lf\n",
         another_string, &x, &y, &z) == 4) {
    /* We're reading coordinates, keep going */
    molecule_base = atom_realloc(molecule_base, ++molecule_size,
                 strlen(another_string) + 1);

    strcpy(molecule_base[molecule_size - 1].label, another_string);
    molecule_base[molecule_size - 1].coordinates[0] = x;
    molecule_base[molecule_size - 1].coordinates[1] = y;
    molecule_base[molecule_size - 1].coordinates[2] = z;

  } else {
    do_loop = false;
  }

  }

  /* Now ... initialize the atomic numbers and last/next links */
  for (i = 0; i < molecule_size; i++) {
  if ( (molecule_base[i].atomic_number =
      atom_lab_to_num(molecule_base[i].label) ) == 0 ) {
    warn_out("Unknown atom label encountered in main, it's most "
         "likely that the structure file is corrupted");
  }

  if ( i != 0 ) {
    molecule_base[i].previous = molecule_base + i - 1;
  }

  if ( i != molecule_size  - 1 ) {
    molecule_base[i].next = molecule_base + i + 1;
  }
  }

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  /* We're done with the Original_structure.raw file, close it and work */
  /* on the Connectivity.raw file */

  if ( fclose( work_file ) == EOF ) {
  warn_out("Unable to close Original_structure.raw in main(), this "
       "should not be fatal, but is definately a problem");
  }

  if ( strlen(db_path) + strlen("Connectivity.raw") + 2 > MAXSTR ) {
  error_exit("Potential string overflow in main()");
  }
```

```
  if (! strcpy(work_string, db_path) ) {
  error_exit("Unable to copy string in main()");
  }

  strcat(work_string, "/Connectivity.raw");

  if ( ! (work_file = fopen(work_string, "r")) ) {
  error_exit("Unable to open Connectivity.raw in main()");
  }

  /* Work file is open, start the initializations */
  do_loop = true;

  while ( do_loop ) {

  if ( (fgets(work_string, MAXSTR, work_file) ) == NULL) { break; }

  /* Read connectivity information */
  if (sscanf(work_string, "%d %d %f\n", &i, &j, &bond_order) == 3 ) {
    atom_connect(molecule_base + i, molecule_base + j, bond_order);
  } else {
    do_loop = false;
  }
  }

  /* And ... close the file */
  fclose(work_file);

  /* We do need to assign qcodes, since that's the purpose of this
   program */
  if ( assign_qcodes(molecule_base) == 0 ) {
    error_exit("Failed to initialize all qcodes in main()");
  }

  /* At this point, we simply assume the information in the database is */
  /* both complete and correct.  We could do a lot of other checking on */
  /* thinks like making sure the valences are full, making sure the */
  /* formal charges are correctly assigned, and so forth.  The job of */
  /* verifying the integrity of the database information falls on */
  /* ../qdb_utilities, not here, though another program with a similar */
  /* structure may be written to verify the actual structures in the */
  /* database */

#ifdef DMALLOC
  dmalloc_verify(0);
#endif

  for (work_atom = molecule_base; work_atom; work_atom = work_atom->next) {
  /* And print out the actual qcode information */
  for (i = 0; i < QDEPTH; i++ ) {
    /* The body of this loop was taking almost verbatim from qdb_check */
    if ( i ) { printf(" "); }
    /* Note the precision specifier .255.  It is */
    /* important to catch every last digit for recording */
    /* in the Qcodes files. */
    printf("%.255Lg", work_atom->qcode[i]);
  }
  printf("\n");
  }


  /* Before we return, free all of our memory --- the system will happily */
  /* do it, but this also checks the free'ing routines for correctness */
  free_molecule(molecule_base);

  /* We finished sucessfully, let the calling program know */
  did_something = yes;

  if (did_something) { return 1; }

  return 0;

}
```

utilities/generate_stereochemical_descriptors.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* This is a utility to output a stereochemical descriptor file for the */
/* quantum chemistry database.  The following is taken from the original */
/* version of the README file in this directory */

/* According to my dmalloc audit of this program, it is completely */
/* leak free */

/*

The following programs and their purpose/function/usage are as follows:

generate_stererochemical_descriptors <database_directory> <method>

  This program will generate the output text for a
  stereochemical descriptor file.  qdb_utilities.pl will call this
  program to verify or add this descriptor to the appropriate
  directory.  <database_directory> is a full path to the database
  entry to be read and ouput prepared for.  Method is manditory as
  well, and is included from the beginning to prepare for "alternate"
  descriptor schemes.  The program will use the functions in
  ../../../general/atom_handling.c for determination of output, and
  if other schemes (CIP descriptors) are added later, this method
  can be changed.  See the comments in that file for specific details.
  For now, method will be required, but ignored, edit the source file
  to change this.

  Other notes:
  The program is picky about the pathname it gets, it will do unknown
  things if the path has a trailing "/"
*/

/* Includes */
#include <stdio.h>
#include <string.h>

#include "../../../general/atom.h"

/* Macro and type definitions */
#ifndef MAXSTR
#define MAXSTR 256
#endif

#ifndef BOOLEAN
#define BOOLEAN
typedef enum {false = 0, no = 0, true = 1, yes = 1} boolean;
#endif

/* Forward prototypes (for functions used from other files).  Note that */
/* we could also include those files headers, but this would require we */
/* fuss with the Makefile (configure.pl) as well.  It seems most sensible */
/* in a monolithic program like this to just declare what we need */
int assign_qcodes(atom *some_atom);


int main (int argc,char *argv[]) {

/* Variable declarations */
char db_path[MAXSTR], work_string[MAXSTR], another_string[MAXSTR];
FILE* work_file;
boolean do_loop, did_something = no;
double x, y, z;
atom *molecule_base = NULL, *work_atom;
int method = 0, molecule_size = 0, i, j;
float bond_order;

/* Initialization of command line variables */
if ( argc != 3 ) {
  printf(
/* Begin Usage message */
"Usage:\n"
"\n"
"%s <database_directory> <method>\n"
"\n"
"Where <database_directory> is a fully qualified path to the parent\n"
  "database directory, and:\n"
"<method> is provided for future compatability with other methods of\n"
"  determining the stereochemical descriptor.  It is currently\n"
"  ignored, but required.\n"
"\n"
"\n", argv[0]);
error_exit("");

}

if (strlen(argv[1]) > MAXSTR - 1) {
  printf("First argument is too long (> %i) characters", MAXSTR);
  error_exit("");
}
strcpy(db_path, argv[1]);

/* And initialize the method */
method = atoi(argv[2]);

/* Initialization of command line is finished */

/* And ... begin our work */
if ( strlen(db_path) + strlen("Original_structure.raw") + 2 > MAXSTR ) {
  error_exit("Potential string overflow in main()");
}

if (! strcpy(work_string, db_path) ) {
  error_exit("Unable to copy string in main()");
}

strcat(work_string, "/Original_structure.raw");

if ( ! (work_file = fopen(work_string, "r")) ) {
  error_exit("Unable to open Original_structure.raw in main()");
}

/* Work file is open, start the initializations */
do_loop = true;

#ifdef DMALLOC
dmalloc_verify(0);
#endif

while (do_loop) {
  if ( ( fgets(work_string, MAXSTR, work_file) ) == NULL ) { break; }
  if (sscanf(work_string, "%[^,],%lf,%lf,%lf\n",
    another_string, &x, &y, &z) == 4) {
```

```
    /* We're reading coordinates, keep going */
    molecule_base = atom_realloc(molecule_base, ++molecule_size,
      strlen(another_string) + 1);

    strcpy(molecule_base[molecule_size - 1].label, another_string);
    molecule_base[molecule_size - 1].coordinates[0] = x;
    molecule_base[molecule_size - 1].coordinates[1] = y;
    molecule_base[molecule_size - 1].coordinates[2] = z;

  } else {
    do_loop = false;
  }

}

/* Now ... initialize the atomic numbers and last/next links */
for (i = 0; i < molecule_size; i++) {
  if ( (molecule_base[i].atomic_number =
    atom_lab_to_num(molecule_base[i].label) ) == 0 ) {
    warn_out("Unknown atom label encountered in main, it's most "
      "likely that the structure file is corrupted");
  }

  if ( i != 0 ) {
    molecule_base[i].previous = molecule_base + i - 1;
  }

  if ( i != molecule_size  - 1 ) {
    molecule_base[i].next = molecule_base + i + 1;
  }
}

#ifdef DMALLOC
dmalloc_verify(0);
#endif

/* We're done with the Original_structure.raw file, close it and work */
/* on the Connectivity.raw file */

if ( fclose( work_file ) == EOF ) {
  warn_out("Unable to close Original_structure.raw in main(), this "
       "should not be fatal, but is definately a problem");
}

if ( strlen(db_path) + strlen("Connectivity.raw") + 2 > MAXSTR ) {
  error_exit("Potential string overflow in main()");
}

if (! strcpy(work_string, db_path) ) {
  error_exit("Unable to copy string in main()");
}

strcat(work_string, "/Connectivity.raw");

if ( ! (work_file = fopen(work_string, "r")) ) {
  error_exit("Unable to open Connectivity.raw in main()");
}

/* Work file is open, start the initializations */
do_loop = true;

while ( do_loop ) {

  if ( (fgets(work_string, MAXSTR, work_file) ) == NULL) { break; }

  /* Read connectivity information */
  if (sscanf(work_string, "%d %d %f\n", &i, &j, &bond_order) == 3 ) {
    atom_connect(molecule_base + i, molecule_base + j, bond_order);
  } else {
    do_loop = false;
  }
}

/* And ... close the file */
fclose(work_file);

/* We do need to assign qcodes, since the prioritization scheme */
/* (currently) uses them */
if ( assign_qcodes(molecule_base) == 0 ) {
    error_exit("Failed to initialize all qcodes in main()");
}



/* At this point, we simply assume the information in the database is */
/* both complete and correct.  We could do a lot of other checking on */
/* thinks like making sure the valences are full, making sure the */
/* formal charges are correctly assigned, and so forth.  The job of */
/* verifying the integrity of the database information falls on */
/* ../qdb_utilities, not here, though another program with a similar */
/* structure may be written to verify the actual structures in the */
/* database */

/* Now ... start the actual work, we need to assign descriptors to all */
/* of the atoms that need it, and then scan through and build a */
/* descriptor list */

for (work_atom = molecule_base; work_atom; work_atom = work_atom->next) {
  assign_q_s_descriptor(work_atom);
}

#ifdef DMALLOC
dmalloc_verify(0);
#endif

for (work_atom = molecule_base; work_atom; work_atom = work_atom->next) {
  if (work_atom->s_descriptor) {
    printf("%d %c\n", get_atom_offset(work_atom),
      work_atom->s_descriptor);
    did_something = yes;
  }
}

/* Before we return, free all of our memory --- the system will happily */
/* do it, but this also checks the free'ing routines for correctness */
free_molecule(molecule_base);


if (did_something) { return 1; }

return 0;
}
```

utilities/check_torsions.pl

```
#!/usr/bin/perl -wT

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This small program will check all torsion directories in the
# database directory provided as it's first argument.  The second
# argument indicates the mode.  If it is called with the second
# argument as 1, it will look for errors and report a list of messages
# separated by newlines.  If it is called with the second argument as
# 2, it will return a list (again, with newline as the separator)
# which contains: (# of torsion directories, # of files in torsion
# directories, total size (in megabytes) of files, and number of
# torsion single point calculations).

# Note that all of the 'traditional' die cases (open or die, etc.) are
# (or should be) implemented as print "<blah>" and die, since
# ../qdb_utilities.pl only captures the stdout stream.

package main;

eval { require 5.6.1 }
  or die <<MESSAGE;
#################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
#################################################################
MESSAGE

use strict;
require '../../../general/clean_environment.pl';
full_env_clean();

# Find out which package we'll be using to implement is_finished(), and
# use it.
require "../../../general/rc_file_handling.pl" or die
  "Unable to require ../../../general/rc_file_handling.pl ... exiting\n";

open ('RCFILE', '<../../.qdb_checkrc') or print
  "Unable to open ../.qdb_checkrc ... exiting\n" and die;

my($ab_initio_program) = &read_scalar('RCFILE', "local_ab_initio_program");
defined($ab_initio_program) or print
  "Unable to initialize the local ab_initio_program from ../../.qdb_checkrc " .
  "... exiting" and die;

close('RCFILE');

# Launder our ab_initio_program before requiring it.

($ab_initio_program) = $ab_initio_program =~ m/([\w]+)/;
require "../../../perl_modules/${ab_initio_program}_functions.pl" or print
  "Unable to require ../../../general/rc_file_handling.pl ... exiting\n"
  and die;

# Do command line checking
die <<MY_MESSAGE
Wrong number of command line arguments.  The usage is as follows:
     $0 <path>
Where <path> is a full path to the directory that contains the log file
from which to extract the charges.  This program will determine the
format of the file from ../../.qdb_checkrc
MY_MESSAGE
  unless ($#ARGV == 1);

# And begin the work

# Verify the integrity of the first argument
die "Invalid path $ARGV[0]"
  unless (
      -e $ARGV[0] &&    # Exists
      -r $ARGV[0] &&    # Is readable
      -d $ARGV[0]       # and is a directory
     );
```

```
# Verify the integrity of the second argument
die "Invalid mode specified, please use either 1 or two"
  unless ($ARGV[1] == 1 or $ARGV[1] == 2 );

my($db_dir) = $ARGV[0];

# Finally, do whatever the mode requests
if ($ARGV[1] == 1) {

  my(@torsion_dirs);
  opendir(MY_DIR, $db_dir) or print "Unable to open directory " .
  "$db_dir for reading.\n" and exit(1);

  @torsion_dirs = grep { /^torsion/ } readdir(MY_DIR);
  closedir(MY_DIR);

  foreach (@torsion_dirs) {
  my($this_dir) = $_;

  open(NRGFILE, "<$db_dir/$this_dir/Energies") or print "Unable to open " .
    "$db_dir/$this_dir/Energies for reading ... exiting\n" and exit(1);

  # As I discovered in development, it's possible for the Energies
  # file to become bloated (if there's an error in the torsion
  # driver).  To mitigate this possibility, we'll read at most 360
  # lines, then simply return with an error.

  my(%angle_nrg) = ();
  my($line_count) = 0;
  while (<NRGFILE>) {
    my($angle, $energy) = split(/[ \t]+/, $_);
    $angle = int($angle + 0.5);

    $line_count++;
    if ($line_count >= 361) {

    print "Energies file $db_dir/$this_dir/Energies is too large " .
          "<360 entries), this indicates an error in the original " .
        "entry of this file (most likely by torsion_driver.pl), " .
        "manually correct this directory, or contact the database " .
        "administrator for additional help\n";

    exit(1);
    }

    # Check to make certain all of the input and output files are there.

    unless (-r "$db_dir/$this_dir/torsion_$angle.com" and
        -T "$db_dir/$this_dir/torsion_$angle.com") {
    print "Missing file: $db_dir/$this_dir/torsion_$angle.com.\n"
    }

    unless (-r "$db_dir/$this_dir/torsion_$angle.log" and
        -T "$db_dir/$this_dir/torsion_$angle.log") {
    print "Missing file: $db_dir/$this_dir/torsion_$angle.log.\n"
    }

    # Verify that the calculations (as given by the log file) is
    # actually finished:
    is_finished("$db_dir/$this_dir/torsion_$angle.log");

    # Add the values to a hash so they can be checked after the files are
    # verified to be here.

    $angle_nrg{$angle} = $energy;

  }

  # It might be desireable (later) to check to make certain there
  # are enough values, as specified in .qdb_checkrc.  For now we
  # will simply make certain all of the input and output files are
  # in place, and assume the directory is fine if they are.
  }


} elsif ($ARGV[1] == 2) {
  # Provide summary information in the list format given in the
  # introductory comment

  my(@torsion_dirs);
  opendir(MY_DIR, $db_dir) or print "Unable to open directory " .
  "$db_dir for reading.\n" and exit(1);

  @torsion_dirs = grep { /^torsion/ } readdir(MY_DIR);
  closedir(MY_DIR);

  my($torsion_dir_count) = $#torsion_dirs + 1;
  my($total_file_count) = 0;
  my($total_size) = 0;
  my($single_point_count) = 0;
  my(@work_list);

  foreach (@torsion_dirs) {
  my($this_dir) = $_;
  $torsion_dir_count++;

  opendir(DIR, "$db_dir/$this_dir");
  my(@log_files) = grep { /^torsion_.+\.log$/ } readdir(DIR);
  my(@com_files) = grep { /^torsion_.+\.com$/ } readdir(DIR);

  open (TMP, "<$db_dir/$this_dir/Energies") or
    die "Unable to open <$db_dir/$this_dir/Energies for reading\n";
  $total_file_count++;
  @work_list = stat(TMP);
  $total_size += $work_list[7] / 1024 / 1024;
  close(TMP);

  $single_point_count += int(($#log_files + $#com_files + 2) / 2);
  $total_file_count += $#log_files + $#com_files + 2;

  foreach (@log_files) {
    open (TMP, "<$db_dir/$this_dir/$_") or
    die "Unable to open $db_dir/$this_dir/$_ for reading";
    @work_list = stat(TMP);
    $total_size += $work_list[7] / 1024 / 1024;
    close(TMP);
  }

  foreach (@com_files) {
    open (TMP, "<$db_dir/$this_dir/$_") or
    die "Unable to open $db_dir/$this_dir/$_ for reading";
    @work_list = stat(TMP);
    $total_size += $work_list[7] / 1024 / 1024;
    close(TMP);
  }
  }

  # And finally, print the return values to be captured by the calling
    # program
  print "$torsion_dir_count\n";
  print "$total_file_count\n";
  print "$total_size\n";
  print "$single_point_count\n";
}

exit(1);

# The module ../perl_modules/g98_functions uses a my_warn() function.  We'll
# provide our own for this utility.
sub my_warn($) {

  my($message) = shift;

  print "Warning!  $message\n";

}
```

utilities/get_chelpg_charges.pl

```
#!/usr/bin/perl -wT

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This small program extracts the chelpg charges from the relevant file.
# It will call out to a function in
# ../../../perl_modules/{local_ab_initio_program}_functions.pl which
# will extract the necessary charges.

package main;

eval { require 5.6.1 }
  or die <<MESSAGE;
#################################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
#################################################################
MESSAGE

use strict;
require '../../../general/clean_environment.pl';
full_env_clean();

# Do command line checking
die <<MY_MESSAGE
Wrong number of command line arguments.  The usage is as follows:
$0 <filename>
Where <filename> is a fully path qualified filename to the log file
from which to extract the charges.  This program will determine the
format of the file from ../../.qdb_checkrc
MY_MESSAGE
  unless ($#ARGV == 0);

require('../../../general/rc_file_handling.pl');

open (RCFILE, '../../.qdb_checkrc') or die
  "Unable to open .qdb_checkrc for reading ... exiting\n";

my($ab_initio_program) = read_scalar('RCFILE', "local_ab_initio_program");
defined($ab_initio_program) or die
    "Unable to find ab_initio_program in .qdb_checkrc file ... exiting\n";

# Launder the program name, since we need it to open the local functions
($ab_initio_program) = $ab_initio_program =~ m%([\w.]+)%;


close(RCFILE);

require("../../../perl_modules/${ab_initio_program}_functions.pl");
```

```
# And begin the work

# Verify the integrity of the first variable
die "Invalid filename $ARGV[0]"
  unless (
    -e $ARGV[0] &&  # Exists
    -r $ARGV[0] &&  # Is readable
    -f $ARGV[0]    # and is a regular file (not a directory)
  );

sub extract_chelpg_charges($);
my(@charge_list) = extract_chelpg_charges($ARGV[0]);

# The final task is to print the list to standard out
print join("\n", @charge_list);
print "\n";

exit(1);
```

runff

fffront.pl

```
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# The program is the GUI for the entire project.  As creating a GUI is
# a pretty low priority for me, it will likely be incomplete, with
# features being added only as needed.  In particular, we have a need
# for a molecule viewer, so that will be implemented first.

package main;

use strict;
use Tk 800.000;

# Other requires

# Forward function declarations
sub init_dialogs($);
sub init_menu($);
sub init_menu_data;
sub finish_menus($);
sub init_functions($);

# Global variables.  These aren't global in the traditional sense,
# functions may need to call them as $main::variablename.


# Begin main program

# Get our parent window, and do the basic configurations
my($top) = MainWindow->new();
$top->title("Ff front end: a simple GUI");

# Create our menu bar object
$top->configure(-menu => my($menu_bar) =
        $top->Menu(-menuitems => init_menu_data)
         );
# Finish configuration of the menus
init_menu($menu_bar);

# Inintialize all dialogues that will be needed
init_dialogs($top);

# Eventually, Initialize all callback functions
init_functions($top);

# And start the event loop
Tk::MainLoop;

# Begin functions

# This function initializes all dialogues that will be needed for the program
sub init_dialogs($) {

  my($top) = shift;

  return;

}

# This function intializes the menu setup for the program.  More
# specifically, it simply returns an anonymous reference to a
# structure that contains all of the information for the menubar that
# should live across the top of the screen.  While this idiom may be
# the most confusing, it has the (strong) advantage of keeping all of
# the menu initialization in a single place, so it should be
# relatively easy to edit
sub init_menu_data {

  [
   # Note that the following idiom does not allow the top level menus
   # to be configured beyond their individual contents.  These will be
   # configured later.
   map ['cascade', $_->[0], -menuitems => $_->[1] ],
   [ '~File',
   [
    [ 'command', '~Exit',  # Each menu item starts with its mode and
                            # the entry specification
    -accelerator => 'Ctrl-q',  # Additional options are specified next
    -command => sub {exit}, ],
   ],
   ], # End File menu specification

   [ '~Edit',
   [
    [ 'command', 'No commands implemented',
    -state => 'disabled',
    -command => sub {},
    ], # Additional menu items go here in other lists
   ],
   ], # End Edit menu specification

   [ '~Tools',
   [
    [ 'command', 'Visualize system',
    -command => sub { system("molren.pl&") },
    ],
   ],
   ], # End Tools menu specification

   [ '~Help',
   [

    [ 'command', 'Help',
    -command => sub { print "No help system implemented\n"; }
    ],

    '',

    [ 'command', 'Version',
    -command => sub { print "No version information ... yet.\n"; },
    ],

    [
     'command', 'About',
     -command => sub { print "No about information yet.\n"; }
    ],

   ],
   ], # End Help menu specification

  ]; # End Outer block

}  # End init_menus

# Since we can only provide a list of lists in init_menus, this
# function is responsible for doing any extra configurations we want
# with the menus
sub init_menu($) {
  my($menubar) = shift;

  # Get references to all of the menu bits we need.
  my(@menus) = my($file_menu, $edit_menu, $tool_menu, $help_menu) =
  (
   $menubar->entrycget('File', -menu),
   $menubar->entrycget('Edit', -menu),
   $menubar->entrycget('Tools', -menu),
   $menubar->entrycget('Help', -menu),
  );

  # Remove tearoffs
  foreach (@menus) {
  $_->configure(-tearoff => 0);
  }

  # If there's anything else we want to do to the menus, we do them here.

  return;
}


# This function initializes all of the callbacks that the main loop
# will need.
sub init_functions($) {


}
```

molren.pl

```
#!/usr/bin/perl -w

# Copyright (C) 2002, Joshua Radke

# This file is part of ffdev.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
```

```
# the Free Software Foundation, version 2.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

# For correspondence, please contact the original author at
# ffdev.sourceforge.net

# This program's basic structure was taken from Mastering Perl/Tk, by
# O'Reilly and associates, chapter 15.  Other basic structure was
# taken from the Perl OpenGL module, available on CPAN.  Additionally,
# I will be using the following conventions: Functions that are
# primarily OpenGL commands will begin with gl_.  These include
# functions to draw the window, compile lists, and such.  Callback
# functions for handling keyboards and mouse will begin with kcb_ and
# mcb_ respectively.  Init functions will begin with init_, and other
# conventions will be noted here as I come up with them.

# Design update, 4-4-02.  It seems I must defend all of my work by
# 6-1-02.  In order to accomplish this deadline, many 'shortcuts' will
# need to be taken within the final steps of the project, this program
# being one of them.  I develop on a 'very good' graphics platform
# (for the time being), and, while many of the setting will remain in
# variables, the program will be 'hard wired' to use what I consider
# to be good default values.  I will continue to strive to make the
# design such that extending the usefulness and flexibility of this
# program will be realtively easy for new developers, of course.

# Ok, admittedly, there is a real problem with the rotations routines.
# After awhile of manipulating the molecule, the mouse response
# becomes unintuitive.  This has to do with the fact that I'm not
# treating the rotation matrix properly.  I will ignore this problem
# for now, as the program does allow me to get whatever veiw I want,
# which is the stated short term goal.

# Uses and includes
use Tk;
use Tk::Dialog;
use Tk::FBox;
use OpenGL qw/:old :all/;
use Cwd;

BEGIN {
  # Since our own modules aren't properly installed, add to the INC
  # list at compile time
  push(@INC, "../perl_modules");
}

use LINALG qw/:basic/;


# Ok, there seems to be some problems with the glu library.  If
# gluQuadricDrawStyle() is called multiple times, we inevitabely get a
# segmentation fault.  This is definately not ideal, but since we'll
# be drawing all of the quadrics with the same style, we'll initialize
# our quadric object here, and simply reuse the same one.  Regardless,
# we will always get a seg fault when we exit, but I don't see that
# there's much that can be done about it.
my $thisobj = gluNewQuadric;
gluQuadricDrawStyle($thisobj, GLU_FILL);
gluQuadricDrawStyle($thisobj, GLU_FILL);
gluQuadricNormals($thisobj, GLU_SMOOTH);

# Forward function definitions

# Global initializations
sub init_mainwindow($);
sub init_opengl_topwindow($);
sub init_gl_vis_callbacks($);
sub init_menu_data;
sub init_menus($);
sub finish_menus($);
sub init_dialogs($);
sub init_keycallbacks($$);

# Primariliy OpenGL routines
sub gl_render_molecule;      # This function is responsible for either
                             # rendereing the molecular system in
                             # question, or the startup image.  It is
                             # also responsible for handling any
                             # lighting or other special effects
sub gl_config_perspective;   # This function is responsible for
                             # handling the projection, or viewing
                             # volume, and preserving the aspect
                             # ratio in the finished window.
sub gl_compile_molecule;     # This function compiles the molecule
                             # into a display list of a single, rigid
                             # object.  It should only need to be
                             # called when we change a frame, or
                             # change the loaded molecule
sub my_swap_buffers;
sub gl_changesize($);
sub myglpMoveResizeWindow($$$$);
sub gl_restore_defaults;

# Other functions
sub popup_die($);
sub popup_error($);
sub popup_message($);

# Utility functions
sub fileopen;

# Input type initializations
sub ffa_init($);

# Event handlers
sub handle_B1_mouse_motion;
sub handle_B3_mouse_motion;
sub handle_shift_B1_mouse_motion;
sub handle_B1_B3_release;

# Chemistry type functions
sub is_valid_atomic_label($);
sub get_vdw_radius($);
sub get_covalent_radius($);
sub get_atomic_color($);

# Math functions
# sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
my $PI = acos(-1);

# Global variables
my $mw;
my $menu_bar;
my $gl_top;
my @frames;  # This list will be the entire system to be rendered.  We
             # don't do anything fancy, like streaming the input - at
             # least for now.  Perhaps later this should be changed.


# Options relevant to the rendering.  We keep them in a global hash to
# make them easily accessible to the various functions
my %glopt;

gl_restore_defaults;

# Options relevant to the GUI are kep in this hash
my %guiopt = (
        userlevel => 6,    # Ranged from 0 - 9, controls how
                           # much information the user gets, and
                           # what options are presented to
                           # them. At 5, the user should stop
                           # getting most informational popup boxes

       );

# The entire rigid molecule will be compiled into a single display
# list.  For some reason, glGenLists(1) was returning 0, which could
# not be used as a display list.  As a result, we'll simply use 1
# (arbitrarily), safe in the knowledge that this is the only display
# list we'll be using.
my $gl_frame = 1;

# Setup the control window, this should include a useable help menu,
# as well as sliders to control rotation and zooming.  We will also
# make many key bindings for keyboard control of zooming, rotation,
# and translation.  We also need mouse bindings in the opengl window
# to do similar controls.  Finally, we'll need the suite of functions
# that hanldes callbacks for various windows events.

$mw = MainWindow->new;
$mw->title('Molren control center');

init_mainwindow($mw);

# Create our menu bar object
$mw->configure(-menu => $menu_bar =
        $mw->Menu(-menuitems => init_menu_data)
         );
# Finish configuration of the menus
init_menus($menu_bar);

# We want the main control window to be just a control bar.  We can
# simply pick a reasonable width and tell the mainwindow that it needs
# no height (since the height is the height of the actual window below
# the menubar.)
$mw->geometry( "250x0" );

# Initialize any commonly used dialogues we might use throughout the
# program, or more complex dialogue boxes
init_dialogs($mw);

# Before we start fussing with the OpenGL window, resize the main
# window so it looks more like a simple menubar

# Initialize the OpenGL rendering menus
$gl_top = init_opengl_topwindow($mw);
$glopt{'glparentwindow'} = $gl_top;

# Initialize any keyboard callbacks we may need
init_keycallbacks($mw, $gl_top);

# While Perl/Tk does fine visibilty change handling for it's own
# windows, the OpenGL window is really just a very bare window indeed.
# As a result, we need to do our own visibility change handling.
init_gl_vis_callbacks($gl_top);

# Make certain the opengl window is in existence before we start
# rendering to it (i.e., this must be called after init_opengl_topwindow())!
#$mw->repeat( 2000 => sub {gl_config_perspective;
#             gl_render_molecule; });
#$mw->repeat(50 => \&gl_compile_molecule);

# And enter the mainloop
MainLoop;

# End main program

####################################################################
#                    Begin Functions
####################################################################

# Initialize the mainwindow.  This will most likely grow to include
# other intializations, as I need them.
sub init_mainwindow($) {

  my $mw = shift;

  $mw->geometry("250x10");
  $mw->raise();
#  $mw->Button(-text => 'Quit', -command => \&exit)->pack;
```

```
  return;

}

# This function intializes the menu setup for the program.  More
# specifically, it simply returns an anonymous reference to a
# structure that contains all of the information for the menubar that
# should live across the top of the screen.  While this idiom may be
# the most confusing, it has the (strong) advantage of keeping all of
# the menu initialization in a single place, so it should be
# relatively easy to edit
sub init_menu_data {
  [ # Begin outer block
   map ['cascade', $_->[0], -menuitems => $_->[1] ],

   [ '~File',
   [
    [ 'command', 'E~xit',  # Each menu item starts with its mode and
                           # the entry specification
    -accelerator => 'Ctrl-q',  # Additional options are specified next
    -command => sub {exit},
    ],
    [
     'command', '~Open File',
     -accelerator => 'Ctrl-o',
     -command => \&fileopen,
    ],
   ],
   ], # End File menu specification

   [ '~Edit',
   [
    [ 'command', 'G~raphics Preferences',
    -accelerator => 'Ctrl-r',
    -command => sub {},
    ], # Additional menu items go here in other lists
   ],
   ], # End Edit menu specification

   [ '~Tools',
   [
    [ 'command', 'Visualize system',
    -command => sub { print "Invalid option for this program\n" },
    ],
   ],
   ], # End Tools menu specification

   [ '~Help',
   [

    [ 'command', 'Help',
    -command => sub { print "No help system implemented\n"; }
    ],

    '', # This is a divider

    [ 'command', 'Version',
    -command => sub { print "No version information ... yet.\n"; },
    ],

    [
     'command', 'About',
     -command => sub { print "No about information yet.\n"; }
    ],

   ],
   ], # End Help menu specification

  ]; # End outer block
}

# This function 'finishes' configuration of the menus.  Any special
# consideration beyond the most basic of specifications must be done
# here.
sub init_menus($) {

  my($menubar) = shift;

  # Get references to all of the menu bits we need.
  my(@menus) = my($file_menu, $edit_menu, $tool_menu, $help_menu) =
  (
   $menubar->entrycget('File', -menu),
   $menubar->entrycget('Edit', -menu),
   $menubar->entrycget('Tools', -menu),
   $menubar->entrycget('Help', -menu),
  );

  # Remove tearoffs
  foreach (@menus) {
  $_->configure(-tearoff => 0);
  }

  # If there's anything else we want to do to the menus, we do them here.

  return;
}

# Initialize here any dialogue boxes that we'll be needing/reusing
# throughout the program, such as error messages and such.
sub init_dialogs($) {

  return;
}

# This subroutine sets up all of the keyboard callbacks for both the
# main (control) window, and the OpenGL rendering window.  Think
# carefully before binding the same things to each window,
# particularly for mouse bindings (which will be in a different function)
sub init_keycallbacks ($$) {
  my $mw = shift;
  my $glwin = shift;

  $mw->bind('<Control-Key-q>' => sub {exit});
  $glwin->bind('<Control-Key-q>' => sub {exit});

  $mw->bind('<Control-Key-o>' => \&fileopen );
  $glwin->bind('<Control-Key-o>' => \&fileopen );

  $mw->bind('<Control-Key-r>' => sub {} );
  $glwin->bind('<Control-Key-r>' => sub {} );

  $glwin->bind('<B1-Motion>' => \&handle_B1_mouse_motion );
  $glwin->bind('<B3-Motion>' => \&handle_B3_mouse_motion );
  $glwin->bind('<Shift-B1-Motion>' => \&handle_shift_B1_mouse_motion );
  $glwin->bind('<B1-ButtonRelease>' => \&handle_B1_B3_release );
  $glwin->bind('<B3-ButtonRelease>' => \&handle_B1_B3_release );


# Temporary convenience bindings
  $mw->bind('<Key-q>' => sub {exit});
  $glwin->bind('<Key-q>' => sub {exit});

  return;
}

sub init_gl_vis_callbacks($) {
  my($glwin) = shift;


  $glwin->bind("<Configure>", \&gl_changesize);

  # The following two event handlers are a bit odd in how they handle
  $glwin->bind("<Visibility>", sub {
         my $tmp = $glopt{'newframe'};
         $glopt{'newframe'} = 0;
         gl_config_perspective;
         $glwin->update;
         $glopt{'newframe'} = $tmp;
         }
        );
  $glwin->bind("<Expose>", sub {
         my $tmp = $glopt{'newframe'};
         $glopt{'newframe'} = 0;
         gl_config_perspective;
         $glwin->update;
         $glopt{'newframe'} = $tmp;
         }
        );
  return;
}

sub init_opengl_topwindow($) {

  my $mw = shift;

  # The Tk gl_top window is incredibly generic.  We may want to try to
  # enforce a minimum size or somesuch to make sure the window is at
  # least reasonable for viewing molecules.  Fortunately, this is
  # unlikely to be a problem, since the windows manager usually does a
  # decent job at this.
  $gl_top = $mw->Toplevel( );

  $gl_top->title('Molren viewport');

  # Make sure the window is created before we ask for the window id.
  $gl_top->waitVisibility;

  # Get the geometry string from the windowing system, so we know how
  # to initialize the embedded openGL window.
  $gl_top->geometry() =~ /^=?(\d+)x(\d+)?(([+-]\d+)([+-]\d+))/ or
  popup_die "Could not retrieve geometry of OpenGL window from the " .
            "windowing system";

  my($width, $height, $screenx, $screeny) = ($1, $2, $3, $4);

  # Later, this window will definately need some flags to double
  # buffer it, and any other openGL options I might know of.  Note
  # that this initialization will _only_ work on Unix-like systems.
  # Unfortunately, we need to use some windows attributes that don't
  # exists across other platforms.  When the OpenGL module is ported
  # to other systems, we can simply use a if/then type construct to
  # support initializations for other systems.  It's _really_ a
  # bummer, but this package hasn't been ported yet, so there is no
  # other option.  I believe I'll be ok with just rgba mode, and
  # double buffering ... I'll have to do some X research if I need
  # anything fancier, I guess.
  glpOpenWindow(
        parent=> hex($gl_top->id),
        width => $width,
        height => $height,
        attributes => [ GLX_RGBA, GLX_DOUBLEBUFFER ]
         );

  $glopt{'glparentx'} = $width;
  $glopt{'glparenty'} = $height;
  $glopt{'glparentaspect'} = $width/$height;

  # In our final initialization version of this, we won't be drawing
  # anything but a background in this window.

  # Set up the lighting parameters, except for the light source, which
  # will be fixed at the camera positions, but to the right some set
  # amount.

  $glopt{'specular'} = [ 1, 1, 1, 1 ];
  $glopt{'shininess'} = [ 80 ];
  $glopt{'lightposition'} = [ 10, -5, 0, 0 ]; # Sets the light a long
                                             # ways to the right of
                                             # the camera.

  # The following should be changed into options later.
  glShadeModel (GL_SMOOTH);
  glMaterialfv_p(GL_FRONT, GL_SPECULAR, @{$glopt{specular}});
  glMaterialfv_p(GL_FRONT, GL_SHININESS, @{$glopt{shininess}});

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
```

```
  glEnable(GL_DEPTH_TEST);

  # And format LIGHT0
  glLightfv_p(GL_LIGHT0, GL_AMBIENT, 0.1, 0.1, 0.1, 1.0);
  glLightfv_p(GL_LIGHT0, GL_DIFFUSE, 0.8, 0.8, 0.8, 1.0);

  gl_config_perspective;    # Sets up the camera
  gl_render_molecule; # Initializes the view

  my_swap_buffers;

  return $gl_top;

}


  # This callback handles resizing of the OpenGL window
  sub gl_changesize($) {

  my $glwin = shift;

  my($width) = $glwin->Width;
  my($height) = $glwin->Height;

  # Note: Unfortunately, the only workaround for the problems with
  # moving and resizing was that the standard glpMoveResizeWindow()
  # just wasn't working.  The my* version of it repeats the resize
  # many times, as it appears to only work intermittently.  I've
  # worked quite a bit on other solutions, and this seems to be the
  # only way that works.
  myglpMoveResizeWindow(0, 0, $width, $height);

  glViewport(0, 0, $width, $height);

  $glopt{'glparentx'} = $width;
  $glopt{'glparenty'} = $height;
  $glopt{'glparentaspect'} = $width/$height;

  $glopt{'newframe'} = 0;
  gl_config_perspective;
  gl_render_molecule;
  my_swap_buffers;
  $glopt{'newframe'} = 1;

  return;
  }


# For some reason, this functions doesn't work consistenly, it does,
# however, seem to work consitently when repeated several times.  $i
# should be customizeable in the GUI.
sub myglpMoveResizeWindow($$$$) {

  my($xborder, $yborder, $width, $height, undef) = @_;
  my $i = 1000;

  for (; $i > 0; $i-- ) {
  glpMoveResizeWindow($xborder, $yborder, $width, $height);
  }
}

# This function is responsible for creating the scene.  Call
# gl_config_perspective to actually show the scene onscreen.
# This function is enclosed in a BEGIN block because I'm using the
# offset and offsetinc to animate the polygon, just so I know it's
# doing something.

BEGIN {

  my $offset = 0;
  my $offsetinc = 0.04;

  sub gl_render_molecule {

  # If we haven't loaded anything yet, do the wandering circle
  # animation ... make this a simple blank screen, or maybe an
  # introductory screen later.
  unless (defined($frames[0][0][1][0])) {
    # Draw offscreen, we'll swap when we're done
    glDrawBuffer(GL_BACK);

    # Begin drawing our scene
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glClearColor(0, 0, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0, 1, 0);
    glBegin(GL_POLYGON);

    $pi =  3.141592654;
    $d2r = $pi / 180.0;
    $nvert = 60;
    $dangle = 360 / $nvert;

    for ($angle = 0; $angle <= 359; $angle += $dangle) {
    $x = cos($angle * $d2r) + $offset;
    $y = sin($angle * $d2r) + (rand(1) - 0.5)/10000000 ;
    glVertex2f($x, $y);
    }
    glEnd;

    # Prepare for next frame if it's appropriate
    if ( $glopt{'newframe'} ) {
    $offset += $offsetinc;
    if ($offset > 1 or $offset < -1) {
      $offsetinc *= -1;
    }
    }

    return;
  }

  # If it wasn't empty, we must have a molecule (and therefore, a
  # compiled molecule list)

      # One of the peculiarities of using OpenGL is that you must do the
      # things you wish to do in a reverse order.  This means that we
      # don't actually build the molecule first, but we position the
      # camera, then build the molecule relative to the new frame of
      # reference.

  glDrawBuffer(GL_BACK);
  glClearColor(0, 0, 0, 1);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

print "Trying to draw my own thing\n";

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;

  # Before we actually call the list, we need to scale and rotate
  # within the modelview.
  glRotatef($glopt{xrot}, 0.0, 1.0, 0.0);
  glRotatef($glopt{yrot}, 1.0, 0.0, 0.0);
  glRotatef($glopt{zrot}, 0.0, 0.0, 1.0);
#  glScalef($glopt{scale}, $glopt{scale}, $glopt{scale});

  glCallList($gl_frame);

  # And place the light.  Note that the light is a bit quirky right
  # now in that is appears to start in a slightly different place
  # than after one has moved the molecule a bit.  I'm not going to
  # worry about this subtlety right now.

  glLoadIdentity;
  glLightfv_p(GL_LIGHT0, GL_POSITION, @{$glopt{lightposition}});

  return;
  }
}

# This function is responsible for handling what the perspective is,
# and creating an appropriate frustrum, or simply doing an
# orthographic projection.
sub gl_config_perspective {

  # Set the projection matrix
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;

  # For any projection, we'll need to make certain that we'll be
  # _not_ messing with the aspect ratio.  OpenGL simply renders it's
  # scene into the window coordinates, without care for what those
  # coordinates are.  It's our job to make the projection volume
  # 'match' the window aspect ratio.  Note that the following aspect
  # ratio is merely 'proof of concept', as we'll also be handling
  # perspective projections in the future.  At any rate, that's how
  # ya do the orthographic ratio preservation.  The Perspective
  # should be simpler yet, as we can use the GLU function
  # gluPerspective.  Note also that this perspective matrix will
  # need to be transformed with the proper handling for the camera
  # (which I think is the same thing).

  # Note: Due to time constraints, the program will _only_ support an
  # orthographic perspective.

  # The following block handles an orthographic perspective
  {
  if (defined($frames[0][0][1][0])) {
    my $aspect = $glopt{'glparentaspect'};
    my $mult = $glopt{maxdim} / $glopt{scale};
    if ($aspect > 1) {
    glOrtho(-$aspect * $mult, $aspect * $mult,
        -$mult, $mult, $glopt{nearclip}, $glopt{farclip});
    } else {
    glOrtho( -$mult, $mult, -$mult/$aspect, $mult/$aspect,
         $glopt{nearclip}, $glopt{farclip});
    }
  } else {
    my $aspect = $glopt{'glparentaspect'};
    if ($aspect > 1) {
    glOrtho(-$aspect, $aspect, -1, 1,
        $glopt{nearclip}, $glopt{farclip});
    } else {
    glOrtho( -1, 1, -1/$aspect, 1/$aspect,
         $glopt{nearclip}, $glopt{farclip});
    }
  }
  }


  gluLookAt( $glopt{camerax}, $glopt{cameray}, $glopt{cameraz},
       $glopt{lookatx}, $glopt{lookaty}, $glopt{lookatz},
       $glopt{upx}, $glopt{upy}, $glopt{upz} );

#print "Just called:\n";
#print  "gluLookAt( $glopt{camerax}, $glopt{cameray}, $glopt{cameraz},".
#"\tCamera posisition\n".
#"          $glopt{lookatx}, $glopt{lookaty}, $glopt{lookatz},".
#"\tLook at position\n".
#"          $glopt{upx}, $glopt{upy}, $glopt{upz} )".
#"\tUp vector\n";

  return;
}

# This function compiles the existing molecule into a display list,
# for usage by gl_render_molecule, which applies all the 'fancy' stuff
# (lighting, fog, etc).  We will also be reusing the same
# gluQuadricObj forever, since the perl/OpenGL gluDeleteQuadric seems
# to cause core dumps, as well as most of the other Quadric
# functions.  See the section near the beginning for details.

BEGIN {

  sub gl_compile_molecule {

  unless (defined($frames[0][0][1][0])) {
```

```
    return;
  }

  glNewList($gl_frame, GL_COMPILE);
  print "Compiling display list\n";

  glMatrixMode(GL_MODELVIEW);

  # Save the current matrix and attributes, in case this object is
  # drawn intermingled with others
  glPushMatrix;
  glPushAttrib(0);

  my ($i, $j);

  # We'll be using a sphere for our quadric object, and the gl
  # utility library will take care of all of the details
  # (normals for lighting, mainly) for us,.

  for ($i = 0; $i <= $#{$frames[$i]}; $i++) { # Each molecule in
                                              # this frame
    for ($j = 1; $j <= $#{$frames[$i][$i]}; $j++) { # Each atom in
                                              # the molecule
    # Finally, we will see all coordinates here

    my($x, $y, $z) = (
             $frames[$glopt{current_frame}][$i][$j][1],
             $frames[$glopt{current_frame}][$i][$j][2],
             $frames[$glopt{current_frame}][$i][$j][3]
            );
    my $radius =
      get_vdw_radius($frames[$glopt{current_frame}][$i][$j][0]) *
      $glopt{atom_size_multiplier} / 10;
    # We have the coordinates, get the color and render the sphere
    glColor3f(
         get_atomic_color($frames[$glopt{current_frame}][$i][$j][0]));
    glMaterialfv_p(GL_FRONT, GL_SPECULAR,
      0.5, 0.5, 0.5, 1.0 );
    glMaterialfv_p(GL_FRONT, GL_AMBIENT,
      get_atomic_color($frames[$glopt{current_frame}][$i][$j][0]), 1.0 );
    glMaterialfv_p(GL_FRONT, GL_DIFFUSE,
      get_atomic_color($frames[$glopt{current_frame}][$i][$j][0]), 1.0 );

    # This took a bit of figgering to work out.  We need put push
    # the current modelview matrix before rendering the next
    # sphere.  If we ever load identity, then we will be unable to
    # effect the overal position of the list later.  Strange, but
    # true, I guess.
    glPushMatrix;
    glTranslatef($x, $y, $z);
    gluSphere($thisobj,$radius,$glopt{detail_level}, $glopt{detail_level});
    glPopMatrix;

    }
  }

  # Restore the previous matrix and attributes.
   glPopAttrib;
  glPopMatrix;

  glEndList;

  return;
  }

}

# This is our own implementation of glutSwapBuffers.  Note that we
# currently only support *NIX type systems, but we can easily have
# this support the windows function SwapBuffers, though a windows
# programmer will need to write the glue function for it.
sub my_swap_buffers {

  glXSwapBuffers();

  return;
}

# The following function is simply a GUI version of die.  It pops up
# a critical error window, then kills the program, when the button is
# pressed.
sub popup_die($) {
  my $message = shift;

  $deathbox = $mw->Dialog(
             -title => 'Critical error',
             -text =>$message .
               '.  This program will terminate when you click ' .
                 'the End Program button',,
             -bitmap => 'error',
             -buttons => ['End Program'],
             -default_button => 'End Program',
            );
  $deathbox->Show;

  exit;
}

# The following function is much like popup_die(), but it's purpose is
# to indicate an error condition, and tell the user that what they
# wanted to be done would not be done.  It then returns control to
# wherever it was before the error occured.
sub popup_error($) {
  my $mw = shift;
  my $message = shift;

  $warnbox = $mw->Dialog(
             -title => 'Cannot complete request',
             -text => $message . ".  Your request cannot ".
                    "be completed.",
             -bitmap => 'warning',
             -buttons => ['Dismiss'],
             -default_button => 'Dismiss',
            );

  $warnbox->Show;

  return;
}

# The following function is much like the other two popup routines,
# but it simply posts the message.  Most of these should only be
# invoked depeding on the $guiopt{userlevel} variable.  Advanced users
# need not see this information.
sub popup_message($) {
  my $message = shift;

  $msgbox = $mw->Dialog(
             -title => 'Please note:',
             -text => $message,
             -bitmap => 'info',
             -buttons => ['Dismiss'],
             -default_button => 'Dismiss',
            );

  $msgbox->Show;

  return;
}

# The following function opens a file, and initializes the system to
# the contents of the file.
sub fileopen {

  my $filename =
  $mw->getOpenFile(
          -defaultextension => ".ffa",
          -initialfile => "bigtest.ffa",
          -title => "Select a data file to render",
          -initialdir => Cwd::cwd(),
          -filetypes        =>
          [['Force Field Animation',     '.ffa'  ],
           ['Text Files',       ['.txt', '.text']],
           ['All Files',        '*',             ],
          ],
         );

  # We have the chosen file to open, make sure it's one of the formats
  # we recognize, and that the file is readable to us, provide error
  # messages for all 'bad cases'.
  unless ( -r $filename) {
  popup_error "$filename is not readable";
  return;
  }

  unless ( -T $filename ) {
  popup_error "$filename is not a text file.  Only text mode " .
            "reading is currently implemented";
  return;
  }

  # Here is where we make certain it is in a format that we know and
  # understand.  The following section is written the way it is so
  # it's easy to add new translators.
  {
  my %ok_formats = (  # This is a hash of references to functions.
                      # The functions are the ones that are
                      # responsible for initializing the system for
                      # that given filename suffix.
           ffa => \&ffa_init,
          );

  my ($prename, $postname) = ( $filename =~ /(.*)\.(.*)/);

  unless ( exists($ok_formats{$postname} ) ) {
    popup_error "$postname is an unknown suffix, the file cannot " .
        "be opened";
    return;
  }

  # Error checking is done, call the appropriate funtion.  If the
  # userlevel is sufficiently low, inform them that it may take
  # awhile, and that neither streaming, nor progress bar are
  # implemented.
  if ($guiopt{userlevel} <= 5) {
    popup_message <<MESSAGE;
Warning: You are about to initialize the data to be rendered.  Depending on the
length of the input file, the operation could take some time.  This program
currently does not support streaming (playing the info as it's read), and
neither does it provide a progress bar so you know how long the operation will
take.  On any modern system (Athlon 800 or greater), the delay should be
minimal.  The rest of the program will be unresponsive to mouse clicks and all
other input for the duration of this initialization.
MESSAGE
  }
  $ok_formats{$postname}($filename);

  }

  # Before leaving, reset the scene to the default view, and re-render
  # it

  gl_restore_defaults;
  gl_compile_molecule;
  gl_config_perspective;
  gl_render_molecule;
  my_swap_buffers;

  # And give the focus back to the OpenGL window.  Note that this is
  # not a 'nice' thing to do, since the window will take focus, even
  # if another program is active at the time.  Unfortunately, the
  # program was losing focus to the OS during this function, and we
  # want the user to immediately be able to keep working with the new
  # structure.
  $gl_top->focusForce;

  return;
}
```

```
# The function simply restores the defaut values for %glopt.
sub gl_restore_defaults {

  # Initialize 'first time through' type variables.  These will not
  # change when we load up a new system, so they should stay the same.
  # Note that these variables may be changed through customizations as
  # the program is run.
  unless (defined($gl_top)) {
    $glopt{glparentwindow} = undef;  # parent window object
  $glopt{glparentx} = 1;          # width of parent window
  $glopt{glparenty} = 1;          # height of parent window
  $glopt{glparentaspect} = 1;     # width/height

  # Sensitivities for mouse dragging and moving options
  $glopt{xyrotatesensitivity} = 10;
  $glopt{zrotatesensitivity} = 10;
  $glopt{xytranslatesensitivity} = 10;
  $glopt{zoomsensitivity} = 10;

  # The next group of options sets the camera position.  Since
  # there's no molecule, we can just let them be something default.
  $glopt{lookatx} = 0;
  $glopt{lookaty} = 0;
  $glopt{lookatz} = 0;
  $glopt{camerax} = 0;
  $glopt{cameray} = 0;
  $glopt{cameraz} = -1;
  $glopt{upx} = 0;
  $glopt{upy} = 1;
  $glopt{upz} = 0;

  # We also need variable to specify the overall scaling and
  # rotation for the molecule.  (We use scaling, since we're using
  # an orthographic projection)
  $glopt{xrot} = 0;
  $glopt{yrot} = 0;
  $glopt{zrot} = 0;
  $glopt{scale} = 1;

  # The next group of options sets the basic defaults for various
  # viewing options
  $glopt{atom_size_multiplier} = 10;
   $glopt{bond_size_multiplier} = 10;
  $glopt{detail_level} = 20;
  $glopt{lighting} = 1;
  $glopt{fog} = 0;
  $glopt{shading} = 'smooth';

  }

  # Reinitializing variables that should be re-done when we reload a
  # new file.

  # Eventually, %glopt should have members that represent all of the
  # other varialbes we think of in viewing.  We will have only one
  # light source, but that position and intensities should be reset
  # when we load a new structure.  We will have a rectangular prism
  # shaped bounding box for the molecule, which should also be reset,
  # though this requires that we search through the existing molecule.
  # We will also need options for orthographic or perspective drawing,
  # lighting (on/off), lighting properties of our atoms and bonds, and
  # other special things we add as we go (fog, fog color, etc.).

  $glopt{newframe} = 1;             # do we render the next frame?
  $glopt{nearclip} = 0;             # near clipping plane location
  $glopt{farclip} = 1000;           # far clipping plane location
  $glopt{current_frame} = 0;        # What frame are we currently on?

  # If we have initialized a molecule, we need to also initialize the
  # values for the bounding box.  The simple test is that we have at
  # least one atom in the first frame

  if (defined($frames[0][0][1][0])) {
  # We have defined at least one molecule, we need to determine a
  # bounding box for it, otherwise, set the bounding box to a simple
  # default.  The following bounding box finding section has _only_
  # been tested on a single molecule in a single frame.  It should
  # be considered completely alpha in quality until we work on more
  # complicated systems.

  $glopt{minx} = $glopt{maxx} =
  $glopt{miny} = $glopt{maxy} =
  $glopt{minz} = $glopt{maxz} = 0;

  my ($i, $j, $k);
  for ($i = 0; $i <= $#frames; $i++) { # Each frame
    for ($j = 0; $j <= $#{$frames[$i]}; $j++) { # Each molecule in
                                                # this frame
    for ($k = 1; $k <= $#{$frames[$i][$j]}; $k++) { # Each atom in
                                                    # the molecule
      # Finally, we will see all coordinates here
      if ( $frames[$i][$j][$k][1] < $glopt{minx}) {
      $glopt{minx} = $frames[$i][$j][$k][1];
      }
      if ( $frames[$i][$j][$k][1] > $glopt{maxx}) {
      $glopt{maxx} = $frames[$i][$j][$k][1];
      }
      if ( $frames[$i][$j][$k][2] < $glopt{miny}) {
      $glopt{miny} = $frames[$i][$j][$k][2];
      }
      if ( $frames[$i][$j][$k][2] > $glopt{maxy}) {
      $glopt{maxy} = $frames[$i][$j][$k][2];
      }
      if ( $frames[$i][$j][$k][3] < $glopt{minz}) {
      $glopt{minz} = $frames[$i][$j][$k][3];
      }
      if ( $frames[$i][$j][$k][3] > $glopt{maxz}) {
      $glopt{maxz} = $frames[$i][$j][$k][3];
      }
    }
    }
  }
  # We need to move all of the atoms so that the bounding box is
  # centered on the origin, since this is what we'll assume for the
  # default camera position.

  my ($xoffset) = ( $glopt{maxx} + $glopt{minx} ) / 2;
  my ($yoffset) = ( $glopt{maxy} + $glopt{miny} ) / 2;
  my ($zoffset) = ( $glopt{maxz} + $glopt{minz} ) / 2;

  $glopt{minx} -= $xoffset;
  $glopt{maxx} -= $xoffset;
  $glopt{miny} -= $yoffset;
    $glopt{maxy} -= $yoffset;
  $glopt{minz} -= $zoffset;
    $glopt{maxz} -= $zoffset;

  for ($i = 0; $i <= $#frames; $i++) { # Each frame
    for ($j = 0; $j <= $#{$frames[$i]}; $j++) { # Each molecule in
                                                # this frame
    for ($k = 1; $k <= $#{$frames[$i][$j]}; $k++) { # Each atom in
                                                    # the molecule
      # Finally, we will see all coordinates here
      $frames[$i][$j][$k][1] -= $xoffset;
      $frames[$i][$j][$k][2] -= $yoffset;
      $frames[$i][$j][$k][3] -= $zoffset;
    }
    }
  }

  # With the size of the bounding box in place, we can finish the
  # camera position initialization

  my ($max_distance) = $glopt{maxx} > $glopt{maxy} ?
    $glopt{maxx} : $glopt{maxy};
  $max_distance = $max_distance > $glopt{maxz} ?
    $max_distance : $glopt{maxz};

  $glopt{maxdim} = $max_distance;
   $glopt{lookatx} = 0;
  $glopt{lookaty} = 0;
  $glopt{lookatz} = 0;
  $glopt{camerax} = 0;
  $glopt{cameray} = 0;
  $glopt{cameraz} = -($max_distance * 2);
  # And to prevent clipping things while zooming, change nearclip
  # and farclip to this as well
  $glopt{nearclip} = -($max_distance * 50);
  $glopt{farclip} = $max_distance * 5;
  $glopt{upx} = 0;
  $glopt{upy} = 1;
  $glopt{upz} = 0;

  } else {
  # Define a default (unrealistic) bounding rectangular prism for
  # the system.
  $glopt{minx} = -1;
  $glopt{maxx} = 1;
  $glopt{miny} = -1;
  $glopt{maxy} = 1;
  $glopt{minz} = -1;
  $glopt{maxz} = 1;
  $glopt{maxdim} = 1;
  }

  return;
}


# The following block of callbacks are invoked upon various button
# events, and are designed to handle moving the molecule around.
# Remember, whenever we manipulate the contents of the arguments to
# gluLookAt(), we need to maintain the fact that the up vector and the
# difference between the camera position and the lookat coordinates
# remain orthoganal.  Also, we should keep the up vector normalized

BEGIN {

  my $active_mode = 'none';
  my $lastx = 0;
  my $thisx = 0;
  my $lasty = 0;
  my $thisy = 0;

  sub handle_B1_mouse_motion {

  # Get our old and new positions set up
  $lastx = $thisx;
  $lasty = $thisy;
  $thisx = $Tk::event->x;
  $thisy = $Tk::event->y;

  # If we just started this mode, wait for another call before doing
  # anything.
  if ($active_mode eq 'none') {
    $active_mode = 'B1';
    return;
  }

  # Get our differences to apply to the camera change
  my $deltax = $thisx - $lastx;
  my $deltay = $thisy - $lasty;

  # Do the rotation.  In order to do the rotation, we need to break
  # the movements into x and y components, and apply them to
  # $glopt{(xrot|yrot)}.  The x component will be that component
  # perpendicular to the up vector, and the y component will be that
  # component paraless to the y vector.

  my (@upvec) = ($glopt{upx}, $glopt{upy}, $glopt{upz});
  # The following is just a 'simple' way to get a different vector
  # than the up vector, so we can get a unit vector normal to the up
  # vector.
  my (@tmpvec) = ($glopt{upy}, $glopt{upx}, 0);
  my (@xunitvec) = v_perp(@upvec, @tmpvec);
  @xunitvec = v_norm(@xunitvec);
  my (@yunitvec) = v_norm(@upvec);
```

```
# And incorporate the mouse sensitivity
@xunitvec = v_scalar_mult($glopt{xyrotatesensitivity} / 20, @xunitvec);
@yunitvec = v_scalar_mult($glopt{xyrotatesensitivity} / 20, @yunitvec);

# And finally, muliply them by the requested amounts.
@xunitvec = v_scalar_mult($deltax, @xunitvec);
@yunitvec = v_scalar_mult($deltay, @yunitvec);

$glopt{xrot} += ( $xunitvec[0] + $yunitvec[0] );
$glopt{yrot} += ( $xunitvec[1] + $yunitvec[1] );


gl_config_perspective;

gl_render_molecule;

my_swap_buffers;


return;
}

sub handle_B3_mouse_motion {

# Get our old and new positions set up
$lastx = $thisx;
$lasty = $thisy;
$thisx = $Tk::event->x;
$thisy = $Tk::event->y;

# If we just started this mode, wait for another call before doing
# anything.
if ($active_mode eq 'none') {
  print "Movement detected, doing nothing\n";
  $active_mode = 'B3';
  return;
}

# Get our differences to apply to the camera change
my $deltax = $thisx - $lastx;
my $deltay = $thisy - $lasty;

# This function acts a bit differently than the simple rotations.
# The up and down (y) mouse movements will zoom (scale) the model,
# and the right and left (x) motions will rotate the model along
# the z axis.  In effect, simply changin the up vector.

# Handle the zoom options first.  We'll simply incorporate the z
# sensitivity into the 'zoom factor'.  The zoom factor is a little
# bit wierd, since it will be multiplied by the current scale
# factor.

# Note that the following scheme for zooming becomes unbearably
# slow, but _only_ after one has zoomed into the molecule (or
# system) by a ridiculous amound.  I'm not going to worry about
# this for now.

my $zoomfactor;

$zoomfactor = $glopt{zoomsensitivity} / 1000;

if ($zoomfactor > 1) {
  # Slow it down just a bit
  $zoomfactor *= 1 / exp($glopt{scale});
}

$glopt{scale} += $deltay * $zoomfactor;


if ($glopt{scale} <= 0) {
  $glopt{scale} = 0.000001;
}

# There's a (small) problem.  With scaling values of less than 1,
# the lighting vectors are no longer normalized, and there are odd
# artifacts in the rendering of the spheres.  We'll instead do our
# scaling by changing our viewing volume.


# Do the rotation.  In order to do the rotation, we simply need to
# convert the mouse movement to radians (with appropriate scaling
# for sensitivity) and calculate the new upvector.

$glopt{zrot} += $deltax / $glopt{zrotatesensitivity} * 10;

gl_config_perspective;

gl_render_molecule;

my_swap_buffers;

return;
}

sub handle_shift_B1_mouse_motion {

print "Handling shift-B1 mouse motion\n";
# Get our old and new positions set up
$lastx = $thisx;
$lasty = $thisy;
$thisx = $Tk::event->x;
$thisy = $Tk::event->y;

# If we just started this mode, wait for another call before doing
# anything.
if ($active_mode eq 'none') {
  print "Movement detected, doing nothing\n";
  $active_mode = 'sB1';
  return;
}

# Get our differences to apply to the camera change
my $deltax = $thisx - $lastx;
  my $deltay = $thisy - $lasty;

  # This function does x and y translation, which we'll need to
  # interpret based on the current up vector.  $deltax refers to the
  # translation perpendicular to the up vector, and $deltay refers
  # to the translation parallel to the up vector.
  my (@upvec) = ($glopt{upx}, $glopt{upy}, $glopt{upz});
  # The following is just a 'simple' way to get a different vector
  # than the up vector, so we can get a unit vector normal to the up
  # vector.
  my (@tmpvec) = ($glopt{upy}, $glopt{upx}, 0);
  my (@xunitvec) = v_perp(@upvec, @tmpvec);
  @xunitvec = v_norm(@xunitvec);
  my (@yunitvec) = v_norm(@upvec);

  # And incorporate the mouse sensitivity
  @xunitvec = v_s_mult($glopt{xyrotatesensitivity} / 50, @xunitvec);
  @yunitvec = v_s_mult($glopt{xyrotatesensitivity} / 50, @yunitvec);

  # And finally, muliply them by the requested amounts.
  @xunitvec = v_s_mult($deltax, @xunitvec);
  @yunitvec = v_s_mult($deltay, @yunitvec);

  $glopt{camerax} += ($xunitvec[0] + $yunitvec[0]);
  $glopt{lookatx} += ($xunitvec[0] + $yunitvec[0]);
  $glopt{cameray} += ($xunitvec[1] + $yunitvec[1]);
  $glopt{lookaty} += ($xunitvec[1] + $yunitvec[1]);

  gl_config_perspective;

  gl_render_molecule;

  my_swap_buffers;

  return;
  }

  sub handle_B1_B3_release {

  $active_mode = 'none';

  return;
  }

}

# For ease of implementation, all of the init functions should follow
# the following format:

# Before reading the file, we need to decide on the structure
# of the data we'll be using.  The data will consist of several
# 'nested' lists, for the heirearchy's involved in creation of the
# overall graphics to display.
#
# @frames.  Each member of this list will be a reference to all the
#           data for a single frame of the animation.  A single system
#           will have only one fram, of course.  This list is global,
#           all of the other lists will be local - and then have
#           references entered into @frames
# @system   Pointed to by members of @frames, contains referances to
#           @molecule(s)
# @molecule $molecule[0] is a reference to the connectivity of the
#           molecule.  $molecule[1..n] are references to lists
#           containing atomic number, and the coordinates.
#
# Without being able to verify this overall structure, access would
# then look like:
# @{$frames[0]} is the same as a @system
# @{$system[0]} is the same as a @molecule
# @{$molecule[0]} is the same as a @connectivity
# @{$molecule[n]} is something like (H, 0, 0, 0)

# A more complex invocation is:
# $frames[0][0][0][0]    Where the individual values are:
#         ^  ^  ^  ^---  Either 0-2 for connectivity, 0-3 for others
#         |  |  |------  0 for connectivity index, 1 through the end
#         |  |           of the molecule for individual atoms.
#         |  |---------  Index of the particular molecule in the frame.
#         |------------  Index of which frame we're in.

# This function exists to initialize the '.ffa' file format.  All
# initialization functions should eventually be rewritten to provide
# 'status reports' for the progress bar, but this is considered an
# 'advanced feature', and will not be implemented until a later time.
sub ffa_init($) {

  my $filename = shift;
  @frames = ();
  my(@system, @molecule, @connectivity, $line, @scratch_list);
  my($frame_count) = 0;
  my($read_mode) = undef;

  open(INFILE, "<$filename") or
  popup_die("Unable to open $filename for reading");

  # First things first, get the number of frames before proceeding.

while ( <INFILE> ) {
  $line = $_;
  chomp($line);

  # And start reading

  unless ($frame_count) {
  if ($line =~ /^FrameCount = (\d+)/) {
    $frame_count = $1;
    next;
  }
  }

  if ($line =~ /^End Input:/) {
  # Do the same thing here that we do when we run into Begin Frame,
  # i.e., push the lists into the right places.
  unshift(@molecule, [ @connectivity ]);
```

```
  push(@system, [ @molecule ]);
  push(@frames, [ @system ]);
  last;
  }

  if ($line =~ /^Begin frame:/) {
  if (defined($read_mode)) {
    unshift(@molecule, [ @connectivity ]);
    push(@system, [ @molecule ]);
    push(@frames, [ @system ]);
    @system = @molecule = @connectivity = undef;
  }
  $read_mode = 'system';
  next;
  }

  if ($line =~ /^Begin system:/) {
  unless ($read_mode eq 'system') {
    popup_die "Bad input to $0 on line $. of $filename.  Was " .
            "expecting Begin system:, instead, the line " .
        "read was $line\n";
  }
  $read_mode = 'molecule';
  next;
  }

  if ($line =~ /^Begin molecule:/) {
  if ($read_mode eq 'molecule') {
    # Do nothing, this is the first molecule in the system
  } elsif ($read_mode eq 'connectivity') {
    # We have encountered another molecule in the same system
    unshift (@molecule, [ @connectivity ]);
    push (@system, [ @molecule ]);
    @molecule = @connectivity = undef;
  } else {
    # This is an error condition, and shouldn't occur
    popup_die "Bad input to $0 on line $. of $filename.  Was " .
            "expecting read_mode to be \'molecule\' or " .
        "\'connectivity\', instead, it was $read_mode.  The line " .
        "read was $line\n";
  }
  $read_mode = 'coordinates';
  next;
  }

  if ( $line =~ /^Begin coordinates:/ ) {
  unless ($read_mode eq 'coordinates') {
    popup_die "Bad input to $0 on line $. of $filename.  Was " .
            "expecting Begin coordinates:, instead, the line " .
        "read was $line\n";
  }
  next;
  }

  if ( $line =~ /^Begin connectivity:/ ) {
  unless ($read_mode eq 'coordinates') {
    popup_die "Bad input to $0 on line $. of $filename.  Was " .
            "expecting Begin coonnectivity:, instead, the line " .
        "read was $line\n";
  }
  $read_mode = 'connectivity';
  next;
  }

  # Finally, read our input
  if ($read_mode eq 'coordinates') {
  # Read coordinates into @molecule
  @scratch_list = split(/,/, $line);
  # Before recording the values, make certain the values are all
  # acceptable
  unless (is_valid_atomic_label($scratch_list[0]) ) {
    popup_die "Invalid atomic lable \"$scratch_list[0]\" encountered in " .
      "initialization, exiting\n";
  }

  my($i);
  for ($i = 1; $i <= 3; $i++) {
    unless ($scratch_list[$i] =~ /^-?[\d]+.?[\d]*$/ ) {
    popup_die "Non numerical value encountered in initialization of " .
        "coordinates, exiting";
    }
  }

  push (@molecule, [ @scratch_list ]);
  next;
  } elsif ($read_mode eq 'connectivity') {
  # Read coordinates into @connectivity
  @scratch_list = undef;
  ($scratch_list[0], $scratch_list[1], undef) = split(/ /, $line, 3);

  # Before recording the values, make certain they are all
  # acceptable
  unless ($scratch_list[0] =~ /^\d+$/ and $scratch_list[1] =~ /^\d+$/ and
        $scratch_list[0] <= $#molecule and
      $scratch_list[1] <= $#molecule) {
    popup_die "Invalid atom numbers found while initializing connectivity";
  }

  push(@connectivity, [ @scratch_list ]);
  next;
  } else {
  # This is an error condition
  popup_die "Bad input to $0 on line $. of $filename.  Was " .
            "expecting read_mode to be \'connectivity\':, instead, " .
        "it was $read_mode.  The current line was was $line\n";
  }

  # This loop is designed to always continue on a next statement, if
  # it falls through, it is a logical error.
  popup_die "Logical error when initializing $0, exiting";
}

# Verify we read in the information correctly, we have only @frames to
# manipulate here.
#   print "The content of frames was: " .
#   join(", ", @frames) . "\n"; @system = @{$frames[0]};
#   print "The content of system was: " .
#   join(", ", @system) . "\n";
#   @molecule = @{$system[0]};
#   print "The content of the first molecule was: "
#   . join(", ", @molecule ) . "\n";
#   print "Or ... if we want to unpack each reference, it's:\n";
#   foreach (@molecule) {
#   if ($_ == $molecule[0]) {
#     foreach (@{$_}) {
#     print join(", ", @{$_}) . "\n";
#     }
#   } else {
#     print join(", ", @{$_}) . "\n";
#   }
#   }

  close(INFILE);
}

# This subroutine should most certainly be extended to include all of
# the elements at some point, but for our purposes, it's good enough.
sub is_valid_atomic_label($) {

  my($label) = shift;
  if ( $label eq 'H' or
     $label eq 'He' or
     $label eq 'Li' or
     $label eq 'Be' or
     $label eq 'B' or
     $label eq 'C' or
     $label eq 'N' or
     $label eq 'O' or
     $label eq 'F' or
     $label eq 'Na' or
     $label eq 'Mg' or
     $label eq 'Al' or
     $label eq 'Si' or
     $label eq 'P' or
     $label eq 'S' or
     $label eq 'Cl' or
     $label eq 'Ar' or
     $label eq 'Br' or
     $label eq 'Kr' or
     $label eq 'I' or
     $label eq 'Xe' ) {
  return 1;
  } else {
  return 0;
  }

}

# The following function is a simple implementation of the information
# published in: A. Bondi (1964) "van der Waals Volumes and Radii"
# J.Phys.Chem. 68, 441-451.  Note that all returned values are given
# in Angstroms.  The table itself was extracted from
# http://www.ccdc.cam.ac.uk/support/csd_doc/volume1/z1c07076.html.
# It is in a BEGIN block so the hash only needs to be initialized once
BEGIN {

  my(%vdw_radius_hash) =
    qw{
      Ag  1.72      Ar  1.88      As  1.85      Au  1.66
      Br  1.85      C   1.70      Cd  1.58      Cl  1.75
      Cu  1.40      F   1.47      Ga  1.87      H   1.20
      He  1.40      Hg  1.55      I   1.98      In  1.93
      K   2.75      Kr  2.02      Li  1.82      Mg  1.73
      N   1.55      Na  2.27      Ne  1.54      Ni  1.63
      O   1.52      P   1.80      Pb  2.02      Pd  1.63
      Pt  1.72      S   1.80      Se  1.90      Si  2.10
      Sn  2.17      Te  2.06      Tl  1.96      U  1.86
      Xe  2.16      Zn  1.39
    };

  sub get_vdw_radius($) {

  my($label) = shift;

  if (exists($vdw_radius_hash{$label})) {
    return $vdw_radius_hash{$label};
  } else {
    return 2.00;
  }
  }
}

# # The information from the following two functions
# # (get_covalent_radius and get_atomic_color) were taken from  the web page
# at:http://www.brunel.ac.uk/depts/chem/ch241s/re_view/append_a.htm.

# Default Atom Radii and Colours The following radii, except the
# valuesfor the alkali metals, were taken from the table on p87 of
# 'InorganicChemistry' by Klienberg, J., Argersinger, W.J., and
# Griswold, E.,D.C. Heath and Co., Boston 1960 which listed the
# covalent radii of allthe elements. In the case of the alkali metals
# their ionic radii wereused rather than their covalent radii, these
# values were taken fromCRC Handbook of Chemistry and Physics, pF209,
# 56th Ed.,Ed. R.C. Weast, CRC Press, Ohio, 1975-76.

BEGIN {

  my( %covalent_radius_hash ) =
  (
   H  =>  .37,  He  =>  1,  Li  =>  .68,
   Be  =>  .889,  B  =>  .8,  C  =>  .771,
   N  =>  .74,  O  =>  .74,  F  =>  .72,
   Na  =>  .97,  Mg  =>  1.364,  Al  =>  1.248,
   Si  =>  1.173,  P  =>  1.1,  S  =>  1.04,
   Cl  =>  .994,  K  =>  1.33,  Ca  =>  1.736,
   Sc  =>  1.439,  Ti  =>  1.324,  V  =>  1.224,
   Cr  =>  1.172,  Mn  =>  1.168,  Fe  =>  1.165,
   Co  =>  1.157,  Ni  =>  1.149,  Cu  =>  1.173,
```

```
 Zn  =>  1.249,  Ga  =>  1.245,  Ge  =>  1.223,
 As  =>  1.21,  Se  =>  1.17,  Br  =>  1.142,
 Rb  =>  1.47,  Sr  =>  1.914,  Y  =>  1.616,
 Zr  =>  1.454,  Nb  =>  1.342,  Mo  =>  1.291,
 Tc  =>  1.25,  Ru  =>  1.241,  Rh  =>  1.247,
 Pd  =>  1.278,  Ag  =>  1.339,  Cd  =>  1.413,
 In  =>  1.497,  Sn  =>  1.412,  Sb  =>  1.41,
 Te  =>  1.37,  I  =>  1.334,  Cs  =>  1.67,
 Ba  =>  1.981,  La  =>  1.69,  Hf  =>  1.442,
 Ta  =>  1.343,  W  =>  1.299,  Re  =>  1.278,
 Os  =>  1.255,  Ir  =>  1.26,  Pt  =>  1.29,
 Au  =>  1.336,  Hg  =>  1.44,  Tl  =>  1.549,
 Pb  =>  1.538,  Bi  =>  1.52,  Po  =>  1.53,
 X  =>  1.6,
);

sub get_covalent_radius($) {

my $atom = shift;

if (exists($covalent_radius_hash{$atom})) {
  return $covalent_radius_hash{$atom};
} else {
  return undef;
}
}

my ( %color_hash ) =
(
 H  =>  [ 1.000, 1.000, 1.000 ],  He  =>  [ 0.784, 0.784, 1.000 ],
 Li  =>  [ 0.784, 0.784, 1.000 ],  Be  =>  [ 0.784, 0.784, 1.000 ],
 B  =>  [ 0.000, 0.667, 0.000 ],  C  =>  [ 0.488, 0.488, 0.488 ],
 N  =>  [ 0.000, 0.000, 0.608 ],  O  =>  [ 0.941, 0.000, 0.000 ],
 F  =>  [ 0.000, 0.686, 0.000 ],  Na  =>  [ 0.588, 0.588, 0.588 ],
 Mg  =>  [ 0.588, 0.588, 0.588 ],  Al  =>  [ 0.941, 0.941, 1.000 ],
 Si  =>  [ 0.863, 0.588, 0.863 ],  P  =>  [ 0.667, 0.333, 0.333 ],
 S  =>  [ 1.000, 1.000, 0.000 ],  Cl  =>  [ 0.392, 0.902, 0.000 ],
 K  =>  [ 0.588, 0.588, 0.588 ],  Ca  =>  [ 0.588, 0.588, 0.588 ],
 Sc  =>  [ 0.588, 0.588, 0.588 ],  Ti  =>  [ 0.588, 0.588, 0.588 ],
 V  =>  [ 0.588, 0.588, 0.588 ],  Cr  =>  [ 0.588, 0.588, 0.588 ],
 Mn  =>  [ 0.588, 0.588, 0.588 ],  Fe  =>  [ 0.588, 0.588, 0.588 ],
 Co  =>  [ 0.588, 0.588, 0.588 ],  Ni  =>  [ 0.588, 0.588, 0.588 ],
 Cu  =>  [ 0.588, 0.588, 0.588 ],  Zn  =>  [ 0.588, 0.588, 0.588 ],
 Ga  =>  [ 0.706, 0.000, 0.706 ],  Ge  =>  [ 0.706, 0.000, 0.706 ],
 As  =>  [ 0.588, 0.588, 0.588 ],  Se  =>  [ 0.608, 0.608, 0.000 ],
 Br  =>  [ 0.502, 0.784, 0.196 ],  Rb  =>  [ 0.588, 0.588, 0.588 ],
 Sr  =>  [ 0.588, 0.588, 0.588 ],  Y  =>  [ 0.588, 0.588, 0.588 ],
 Zr  =>  [ 0.588, 0.588, 0.588 ],  Nb  =>  [ 0.588, 0.588, 0.588 ],
 Mo  =>  [ 0.588, 0.588, 0.588 ],  Tc  =>  [ 0.588, 0.588, 0.588 ],
 Ru  =>  [ 0.588, 0.588, 0.588 ],  Rh  =>  [ 0.588, 0.588, 0.588 ],
 Pd  =>  [ 0.588, 0.588, 0.588 ],  Ag  =>  [ 0.588, 0.588, 0.588 ],
 Cd  =>  [ 0.588, 0.588, 0.588 ],  In  =>  [ 0.588, 0.588, 0.588 ],
 Sn  =>  [ 0.588, 0.588, 0.706 ],  Sb  =>  [ 0.588, 0.588, 0.706 ],
 Te  =>  [ 0.588, 0.588, 0.706 ],  I  =>  [ 1.000, 0.000, 1.000 ],
 Cs  =>  [ 0.588, 0.588, 0.588 ],  Ba  =>  [ 0.588, 0.588, 0.706 ],
 La  =>  [ 0.588, 0.588, 0.706 ],  Hf  =>  [ 0.588, 0.588, 0.706 ],
 Ta  =>  [ 0.588, 0.588, 0.706 ],  W  =>  [ 0.588, 0.588, 0.706 ],
 Re  =>  [ 0.588, 0.588, 0.706 ],  Os  =>  [ 0.588, 0.588, 0.706 ],
 Ir  =>  [ 0.588, 0.588, 0.706 ],  Pt  =>  [ 0.588, 0.588, 0.706 ],
 Au  =>  [ 0.588, 0.588, 0.706 ],  Hg  =>  [ 0.588, 0.588, 0.706 ],
 Tl  =>  [ 0.588, 0.588, 0.588 ],  Pb  =>  [ 0.588, 0.588, 0.706 ],
 Bi  =>  [ 0.588, 0.588, 0.588 ],  Po  =>  [ 0.588, 0.588, 0.588 ],
 X  =>  [ 0.392, 0.196, 1.000 ],
);

sub get_atomic_color($) {

my $atom = shift;

if (exists($color_hash{$atom})) {
  return ( $color_hash{$atom}[0],
       $color_hash{$atom}[1],
       $color_hash{$atom}[2]
     );
} else {
  return undef;
}

return;
}
}
```

shlib/CFUNCS

Makefile.PL

```
use ExtUtils::MakeMaker;
eval { require 5.6.1 }
  or die <<EOD;
############################################################
### This module has been shown to not compile on perl 5.003 and 5.004.
### Also note that 5.6.0 has a bug which makes loading of user
### installed modules not work.  Please upgrade your perl to at least
### 5.6.1 before trying to use this extension.  See
### "http://www.perl.com/pub/language/info/software.html" for
### information
############################################################
EOD

# The PREFIX and MYEXTLIB variables are passed into subdirectories,
# so using relative directories results in a nonsensical install.
# Since we assume we'll always be running this from this directory,
# we simply use the following to set the install dierectories.
use Cwd;
$my_install_dir = Cwd::getcwd() . '/../cpmodule';

# Symbolic links do not copy well between systems (*DOH*!), so we'll
# have this file simply install copies of the necessary files.
use File::Copy;
copy("../../qdb/qdb_shared_functions.c",
  "./modlib/qdb_shared_functions.c");
copy("../../qdb/qdb_shared_functions.h",
  "./modlib/qdb_shared_functions.h");

# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    'NAME'    => 'CFUNCS',
    'VERSION_FROM'  => 'CFUNCS.pm', # finds $VERSION
    'PREREQ_PM'    => {}, # e.g., Module::Name => 1.1
    'LIBS'    => ['-lm'], # e.g., '-lm'
    'DEFINE'    => '', # e.g., '-DHAVE_SOMETHING'
    'INC'    => '', # e.g., '-I/usr/include/other'
  'MYEXTLIB'  =>  'modlib/libmylib$(LIB_EXT)',
  'PREFIX'  =>  $my_install_dir,
  'LIB'  =>    $my_install_dir
);

sub MY::postamble {
'
$(MYEXTLIB): modlib/Makefile
  cd modlib && $(MAKE) $(PASSTHRU)
';
}
```

CFUNCS.pm

```
package CFUNCS;

require 5.005_62;
use strict;
use warnings;
use Carp;

require Exporter;
require DynaLoader;
use AutoLoader;

our @ISA = qw(Exporter DynaLoader);

# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.

# This allows declaration  use CFUNCS ':all';
# If you do not need this, moving things directly into @EXPORT or @EXPORT_OK
# will save memory.

# The Perl side will not need MAXSTR and QDEPTH, so they're not exported.
our %EXPORT_TAGS = ( 'all' => [ qw(
) ] );

our @EXPORT_OK = ( @{ $EXPORT_TAGS{'all'} } );

our @EXPORT = qw(
);
our $VERSION = '0.01';

sub AUTOLOAD {
    # This AUTOLOAD is used to 'autoload' constants from the constant()
    # XS function.  If a constant is not found then control is passed
    # to the AUTOLOAD in AutoLoader.

    my $constname;
    our $AUTOLOAD;
    ($constname = $AUTOLOAD) =~ s/.*:://;
    croak "& not defined" if $constname eq 'constant';
    my $val = constant($constname, @_ ? $_[0] : 0);
    if ($! != 0) {
  if ($! =~ /Invalid/ || $!{EINVAL}) {
      $AutoLoader::AUTOLOAD = $AUTOLOAD;
      goto &AutoLoader::AUTOLOAD;
  }
  else {
      croak "Your vendor has not defined CFUNCS macro $constname";
  }
    }
    {
  no strict 'refs';
  # Fixed between 5.005_53 and 5.005_61
  if ($] >= 5.00561) {
      *$AUTOLOAD = sub () { $val };
  }
  else {
      *$AUTOLOAD = sub { $val };
  }
    }
    goto &$AUTOLOAD;
}

bootstrap CFUNCS $VERSION;

# Preloaded methods go here.

# Autoload methods go after =cut, and are processed by the autosplit program.

1;
__END__
```

```
# Below is stub documentation for your module. You better edit it!

=head1 NAME

CFUNCS - Perl extension to use C code already in use for the ff project

=head1 SYNOPSIS

  use CFUNCS;

  Firstly, I can't guarantee that this will be kept up to date, if you
  find this documentation in say, 2004, and are wondering more about it,
  look for me ... wherever I'll be (Joshua Radke)

  float get_qcode_deviance(\@qcode1, \@qcode2, $length_to_compare);

  Also, since this c function called another c function, it needed to
  be included:
  warn_out($message);
  This simply prints a message to std_out, it may be written later to
  put the message elsewhere.

=head1 DESCRIPTION

Stub documentation for CFUNCS, created by h2xs. It looks like the
author of the extension was negligent enough to leave the stub
unedited.

Yeah, yeah, sure I didn't.  The (short) descriptions of what the functions
do are in the previous section, look there.

=head2 EXPORT

None by default.

=head2 Exportable constants

  None of the constants (yet) are exported, since they're only needed
  by the C side.


=head1 AUTHOR

Joshua Radke, radke@kilauea.colorado.edu

=head1 SEE ALSO

perl(1), perlxs(1), perlxstut(1), perlapi(1).

=cut
```

typemap

```
# Before `make install' is performed this script should be runnable with
# `make test'. After `make install' it should work as `perl test.pl'

######################### We start with some black magic to print on failure.

# Change 1..1 below to 1..last_test_to_print .
# (It may become useful if the test is moved to ./t subdirectory.)

BEGIN { $| = 1; print "1..1\n"; }
END {print "not ok 1\n" unless $loaded;}
use CFUNCS;
$loaded = 1;
print "ok 1\n";

######################### End of black magic.

# Insert your test code below (better if it prints "ok 13"
# (correspondingly "not ok 13") depending on the success of chunk 13
# of the test code):

@a = (3, 2, 9, 4, 5, 6, 12, 1, 1, 1, 1, 1, 1, 1);
@b = (3, 2, 9, 4.1, 6, 9);

# while (1) {
$answer = CFUNCS::get_qcode_deviance(\@a, \@b);
# }
# $answer = CFUNCS::get_qcode_deviance(\@a, \@b, 6);

print "Answer = $answer\n";
```

modlib/qdb_shared_functions.h

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

/* Includes */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <signal.h>


/* The following is a memory debugging library */
#ifdef DMALLOC
#include <dmalloc.h>
#else
/* Dmalloc has it's own string library */
#include <string.h>
#endif

/* Defines */

#ifndef MAXSTR
#define MAXSTR 256
#endif

#ifndef QDEPTH
#define QDEPTH 20 /* Important!  This is also defined in assign_qcodes.c, */
          /* which should _not_ be dependant on the qdb code, */
          /* (which is why this header isn't included in it) */
#endif

#ifdef __unix__
/* Put UNIX specific defines here.  Note that directory access will */
/* definately differ from system to system */
#endif

/* Prototypes */
void warn_out(char *message);
float get_qcode_deviance(long double *qcode1, long double *qcode2,
  int length);
```

modlib/qdb_shared_functions.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include "qdb_shared_functions.h"

/* This is another standard function, placed in a 'standard' place */
void error_exit(char *message) {
  printf("%s ... exiting.\n", message);
  exit(0);
}

/* This is a standard small warning function.  It's in the shared */
/* functions because get_qcode_deviance() calls it */
void warn_out(char *message) {
  printf("Warning:  %s\n", message);

  return;
}

/* This function takes two vectors and a desired tolerance, and compares */
/* them.  It returns a float which is the deviance, a custom comparison */
/* function that can be changed as needed */

float get_qcode_deviance(long double *qcode1, long double *qcode2,
  int length) {

const long double slop = 0.000000000001;
int i, exact_match;
float weighting_factor, fractional_match;
double sum, diff, temp_double;

/* Error checking */
if (qcode1 == NULL) {
  warn_out("first atom passed to is_qcode match is NULL, this was most "
      "likely unintended, but should not be fatal");
}
if (qcode2 == NULL) {
  warn_out("second atom passed to is_qcode match is NULL, this was most "
```

```
        "likely unintended, but should not be fatal");
}

/* Test for exact part.  If the qcodes were generated on the same machine, */
/* they will be exact, otherwise, the slop may need to be adjusted */
for ( i = 0; i < length &&
  fabs(qcode1[i] - qcode2[i]) < slop; i++ ) {
}

exact_match = i;

/* And now we test the fractional part */
/* Note that this is a weighted squares.  The most important terms of the */
/* qcode vector will be the first few, and the terms near the end are */
/* ultimately of much less importance.  Therefore, we use an exponential */
/* weighting to emphasize the first terms more */
weighting_factor = 0.5;
sum = 0;
for ( i = exact_match; i < length; i++ ) {
  diff = exp( -1.0 * weighting_factor * (i - exact_match + 1) ) *
    fabs(qcode1[i] - qcode2[i]);
  sum += diff * diff;
}

/* And we need an _average_ squares difference, since this may be done */
/* with different values of length, and we want them to be comparable */
sum /= length - exact_match;
/* if sum is currently 0, we don't want to take the log of it */

/* Note here that we're defining the value 0.999 to be the 'practical */
/* perfect' match.  This could be checked for later if the calling */
/* routine cares */
if (sum == 0.0 ) {return ((float)exact_match + 0.999) ;}
temp_double = log(sqrt(sum));
sum = temp_double < 0 ? -1 * temp_double : 0;

/* Recall that fractional_match checks the match of ites beyond what we */
/* demanded an exact match for.  Empirically, if sum = 6 it is a very bad */
/* match, 10 is really quite good, 15 is excellent, 20 is nearly perfect, */
/* and 25 is a practical maximum.  As a result, to make the input more */
/* 'intuitive' for users of the program, the number they provide after the */
/* decimal point will be in the form of a percent, and the sum just */
/* calculated will be multiplied by four.  For translation then, 20 is a */
/* totally crappy match, while 40% is quite good, and 60% is about perfect */

fractional_match = sum >= 25 ? 0.999 : sum * 4 / 100;

return (float)exact_match + fractional_match;

}
```

modlib/Makefile.pl

```
use ExtUtils::MakeMaker
$Verbose = 1;
WriteMakefile(
  'NAME'  =>  'CFUNCS::modlib',
  'LIBS'  =>  ['-lm'],
  'clean'  =>  {'FILES'  =>  'libmylib$(LIBEEXT)'},
);

sub MY::top_targets {
'
all :: dynamic
pure_all :: dynamic
dynamic ::  libmylib$(LIB_EXT)
libmylib$(LIB_EXT): $(O_FILES)
  $(AR) -cr libmylib$(LIB_EXT) $(O_FILES)
  $(RANLIB) libmylib$(LIB_EXT)
  cp qdb_shared_functions.o modlib.o
';
}
```

sim

nrgforce.h

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */


/* This header file contains all of the information needed by the
   functions in its corresponding c file.  Before starting up with the
   definitions, however, we need to establish some ground rules.  The
   nrgforce functions are designed to be configured at run time, and
   as such, we needed to decide on a univeral function prototype, so
   they can be executed with function pointers.  (as a side note, it
   would most certainly have been more straightforward to write this
   code in C++, but we chose not to exercise this option for several
   reasons.  First and foremost, oo programming, while slowly gaining
   popularity in chemistry and physics, is simply too far away from
   the way these scientists (including the developers!) think.  Since
   an overall design goal of the project is to be able to entice
   scientists to further develop this code base, an oo language is
   absolutely opposed to this notion.  Secondly, since these functions
   are to be used in very computationally intensive applications, the
   (admittedly small) loss of efficiency in runtime was another
   problem.

   With that out of the way, we can go on to discuss the universal
   energy function prototype.  It is as follows:

   long double function_name(atom **atoms_to_compute, long FLAGS, double*
     optional_list_of_values, global_sim_parms* parms);

   The function name is traditionally chosen to reflect the purpose of
   the function as follows (<>'s are normally in all function
   prototype names, []'s represent parts of the name that may or may
   not be there):

   <type>_[variant]_<force_field_name>_[version]();

   Where:

     type            = primary source of energy term.  Current
                       standard names include: bond (stretching),
         bend (of bonds), torsion (proper), inv
         (inversion,  or umbrella bending), vdw (van
         der waals), and coul (coulumbic forces).

   variant         = if there is a common name for the variant of
                       that portion of a force file, it is noted in
         the function name here.

   force_field_name = This is the 'proper' name of the force field
                       or method used to implement this particular
         energy and force evaluation function.
         Examples might include dreiding, dreiding2,
         OLPS, and AMBER.  If a particular group has
         made some type of hybrid force field
         component, this section is named after that
         (see the bouldergroup functions).

   version         = The version portion of the name is for either
                       the actual functional implementation, or
         'unamed' variants.  In some implementations,
         one algorithm may be faster than another, and
         this mechanism provides the functionality to
         implement idependant functions.  Additionally,
         particularly in the case of 'custom hybrid'
         force field components, there may be several
         variants of a particular evaluation.

   Further details of the function prototype:

   As mentioned previously, the prototype is;

   function_name(atom **atoms_to_compute, long FLAGS, double*
     optional_list_of_values, global_sim_parms* parms);

   atoms_to_compute is normally a list of all of the molecules in the
   simulation, but the user (developer) has the freedom of evaluating
   the energy and/or force for a smaller part of the system if that is
   desired.

   FLAGS is a single long int with boolean options bitwise or'd into
   it.  Because of this choice, the functions will not run on systems
   that do not have at least 64 bit long int's.  This should be
   checked at startup, but is not currently implemented.  Once again,
   a future project.  Please note that FLAGS is declared as a long
   int, not an unsigned long int.  This is because implementation of
   long int's should be system independant, whereas unsigned types may
   vary from system to system.  The first 10 places of the int are
   reserved for 'global' types of parameters.  Other places are
   reserved for the function's own usage.  The current global options
   are:

   0     RETURN_ENERGY     Calculate energy (and return it)
   1     UPDATE_FORCES     Update forces
   2     INIT_NRG_FUNC     Initialize the static members of the energy
                           evaluation function.

   optional_list_of_values is a list of double's that may need to be
   incorporated into the funtion.  For example, we may choose to use
   different values for the dreiding2 bond stretches, but still want
   to use the same protocol (all bonds of a given order have the same
   constants).  This can be accomplished by providing a flag as well
   as the additional values (though this will not be incorporated into
   the original implementation of the dreiding2 functions).  Note that
   function design will be most efficient when these values are
   initialized once (into static variables), and left alone for the
   duration of the simulation.

   global_sim_parms* parms is a single data structure that is
   initialized in the beginning of the simulation, and for the most
   part, should never change.  It is provided to hold any values that
   have global effects on the simulation (such as periodic boundary
   conditions).  Note that it should not become a junkyard for
   variables that can just as easily be supplied to the function
   directly.  Its exact contents are bound to evolve with time.
```

```
   This is the end of the background for this library */

/* Includes */
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include "../general/atom.h"

/* Conditional includes */
#ifndef VECTOR_H
#define VECTOR_H
#include "../general/vector.h"
#endif

/* The following is a memory debugging library */
#ifdef DMALLOC
#include <dmalloc.h>
#else
/* Dmalloc has it's own string library */
#include <string.h>
#endif

/* Defines */
#define MAX_TERMS 32  /* The maximum nuber of summed terms in a force
            field.  Note that the implementation would be
            better off if this was simply decided at
            runtime, and it may be fixed later.  */
#define MAX_FLINE 1024 /* The maximum number of characters on a line in
             a file.  It must be larger, since qcodes can
             be quite long */

/* Note an oddity with the usage of INIT_NRG_FUNC (as defined a couple
   of lines ahead).  Unless UPDATE_FORCES is also defined when
   INIT_NRG_FUNC is, the molecule will have no forces attatched to the
   atoms.  If any initialization function actually defines
   UPDATE_FORCES, it will be that function's responsibility to
   initialize it.   Note also that the library has no capabilities for
   freeing the space it allocates.  This will never be a problem,
   unless a program performs multiple initializatons of the force
   field.  If a need for this behavior ever arises, these capabilities
   will need to be included.  */

#define RETURN_ENERGY 1
#define UPDATE_FORCES 2
#define INIT_NRG_FUNC 4   /* Only initialize static members if this
              flag is set */

#ifndef MAX_STR
#define MAX_STR 256
#endif

/* Sloppy coding left me with two versions of this define, clean this
   up someday */
#ifndef MAXSTR
#define MAXSTR 256
#endif

#ifndef BOOLEAN
#define BOOLEAN
typedef enum {false = 0, no = 0, true = 1, yes = 1} boolean;
#endif

/* Typedefs and structs */

/* This structure can contain any global information potentially
   needed by the energy evaluation functions.  Please see the
   introduction to the energy file for details on this */

typedef struct global_sim_parms {
  boolean periodic_boundary_conditions;
} global_sim_parms;

typedef long double (*nrgfunc)(atom **, long, double*, global_sim_parms*);
/* nrgfunc is now a normal data type */

/* The following typedefs are definately the trickiest part of the
   system.  They allow us to use the 'other' pointer in atoms to
   access additional information the atoms carry around with them. */

#define FORCE_VECTOR_INDEX 0
#define VELOCITY_LIST_INDEX 1

#define BOND_LIST_INDEX 2
#define CONN other[BOND_LIST_INDEX]
#define CONNV(this_atom, index) (int)((int *) \
( (this_atom)->other)[BOND_LIST_INDEX])[index]

#define BOND_SPECIAL1_INDEX 3
#define BOS1I other[BOND_SPECIAL1_INDEX]
#define BOS1IV(this_atom, index) (double)((double *) \
( (this_atom)->other)[BOND_SPECIAL1_INDEX])[index]

#define BEND_LIST_INDEX 4
#define ANGLE_LIST other[BEND_LIST_INDEX]
#define ANGLEV(this_atom, index) (int)((int *) \
( (this_atom)->other)[BEND_LIST_INDEX])[index]

/* A float is sufficient for equilibrium bond angles, as other errors
   should end up being larger.  */
#define BEND_SPECIAL1_INDEX 5
#define BES1I other[BEND_SPECIAL1_INDEX]
#define BES1IV(this_atom, index) (float)((float *) \
( (this_atom)->other)[BEND_SPECIAL1_INDEX])[index]

#define INV_LIST_INDEX 6
#define INV_SPECIAL1_INDEX 7

#define TORSION_LIST_INDEX 8
#define TORSION_SPECIAL1_INDEX 9
#define VDW_LIST_INDEX 10
#define VDW_SPECIAL1_INDEX 11
#define COUL_LIST_INDEX 12
#define COUL_SPECIAL1_INDEX 13
#define NEIGHBOR_LIST_INDEX 14

/* This last value is the size that all other pointers will be
   initialized to in init_ff.  It should be equal to the largest of
   the list indices. */
#define MAX_OTHER_INDEX 14

/*******************************************************************/

/* Functions in nrgforce.c */

/* Utility function for external users nrgforce.c */
boolean init_ff(const char* ff_filename, atom **molecule_list);

/* Energy/system evaluation functions */

long double get_system_energy( atom **atom_list, long flags);

/* Energy functions in nrgforce.c */

long double bond_gen_dreiding2(atom **atom_list, long flags,
             double* override, global_sim_parms* parms);


long double bend_gen_dreiding2(atom **atom_list, long flags,
             double* override, global_sim_parms* parms);

long double torsion_bouldergroup1(atom **atom_list, long flags,
             double* override, global_sim_parms* parms);

long double inv_gen_dreiding2(atom **atom_list, long flags,
             double* override, global_sim_parms* parms);

long double vdw_bouldergroup1(atom **atom_list, long flags,
             double* override, global_sim_parms* parms);
/* Flag defs */
#ifndef OMIT12
#define OMIT12     1024   /* 2 ^ 10 */
#endif
#ifndef OMIT13
#define OMIT13     2048   /* 2 ^ 11 */
#endif
#ifndef OMIT14
#define OMIT14     4096   /* 2 ^ 12 */
#endif

long double coul_rawsum(atom **atom_list, long flags,
             double* override, global_sim_parms* parms);
/* This function uses the same flags as vdw_bouldergroup1() */


/* End function definitions */


/* The following information used to be at the end of the .ff_form
   configuration files, but it makes a heck of a lot more sense to
   keep them in a file whose purpose is to be static.  This will be
   the proper location, and should be kept current */

/* Begin descriptions

This section describes (and documents briefly) all available energy
evaluation functions.  If a new energy evaluation function is
written, it should be documented here.  Additional documentation can
be found at the beginning of the actual function, found in
nrgforce.c.  Note that this the 'most current' version of this can be
found at the end of nrgforce.h.  If we create a 'configuration file
builder', it may end up reading the end of the header file, and
including this information there.

global()
options: none

bond_gen_dreiding2()
options: none

bend_gen_dreiding2()
  options: none

torsion_bouldergroup_1()
  options: none

inv_gen_dreiding2()
  options: none

vdw_bouldergroup_1(OMIT12, OMIT13, OMIT14)
  options:  OMIT12  Omit 1,2 non-bonded interactions
        OMIT13  Omit 1,3 non-bonded interactions
        OMIT14  Omit 1,4 non-bonded interactions

coul_rawsum(OMIT12, OMIT13, OMIT14)
  options:  OMIT12  Omit 1,2 non-bonded interactions
        OMIT13  Omit 1,3 non-bonded interactions
        OMIT14  Omit 1,4 non-bonded interactions

End descriptions
*/
```

nrgforce.c

```
/* Copyright (C) 2002, Joshua Radke

This file is part of ffdev.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 2.
```

```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For correspondence, please contact the original author at
ffdev.sourceforge.net */

#include "nrgforce.h"

/* This is the primary library for evaluating energy and forces.  It
   should have a function definition for each function that is a
   _possible_ member of an overall force field. */

/* Global variables */
static nrgfunc funclist[MAX_TERMS];
static global_sim_parms simulation_parameters;
static global_sim_parms* parms = &simulation_parameters;
/* IMPORTANT NOTE!  These are the _only_ global variables that this
   module should need.  Add to these at your own risk! */

/* Static function definitions.  Note that these should _not_ be in in
   a header file, since static functions must be defined in the same
   file they're declared in (and are likewise inaccesible to other
   files)  */
static char* get_full_command(FILE *infile, char *initial_line);
static long int get_init_flags(char *command);
static void initialize_forces_and_velocities(atom *this_molecule);

/* To keep compiler happy (parms is not used in it's initial
   implementation, other than simply defining it */
static void no_op_blank_function() {
  parms = NULL;
  return;
  no_op_blank_function();
}

boolean init_ff(const char* ff_filename, atom **molecule_list) {

  FILE *cfg_file;
  int term_index = 0;
  char line[MAX_FLINE];
  char *line_p;
  int member_index;

  /* The first task is to re-align the molecule list, so the members
   all point to the base of the molecule they represent */
  member_index = 0;
  while ( molecule_list[member_index] ) {
  while( molecule_list[member_index]->previous != NULL ) {
    molecule_list[member_index] = molecule_list[member_index]->previous;
  }
  member_index++;
  }

  /* Before processing the input file, be certain to initialize any of
   the global_sim_parms default values, in case none are provided in
   the configuration file */

  /* Before processing the input file, initialize space on all other's
   for usage by the initialization functions.  The original design
   only changed the size of the other pointers when asked, but
   keeping an index of the current size adds overhead, and makes
   maintenance a bit more challenging.  It assumes that the other
   pointers are uninitialized.  If they are not uninitialized,
   initialization of the force field makes no sense. */
  member_index = 0;
  while ( molecule_list[member_index] != NULL ) {
  atom *this_atom;
  int i;

  for ( this_atom = molecule_return_base(molecule_list[member_index]);
      this_atom != NULL; this_atom = this_atom->next ) {
    if ( this_atom->other != NULL ) {
    error_exit("Non-NULL other pointer found in initial molecules "
          "in init_ff()");
    }

    if ( ( this_atom->other =
       malloc( (MAX_OTHER_INDEX + 1) * sizeof(void *) ) ) == NULL ) {
    error_exit("Unable to allocate memory for other array in init_ff()");
    }

    /* Finally, set all the pointers of this array to NULL */
    for (i = 0; i < MAX_OTHER_INDEX; i++) {
    this_atom->other[i] = NULL;
    }
  }
  member_index++;
  }

  cfg_file = fopen(ff_filename, "r");

  if ( cfg_file == NULL ) {
  return no;
  }

  line_p = fgets(line, MAX_FLINE, cfg_file);

  if (line_p == NULL) {
  return 0;
  }

  fprintf(stderr, "Initializing force field ... \n");

  while (line_p != NULL) {
  /* Loop local variables */
  int i;
  long init_flags;
  /* When a function implements the override portion, it must be
     handled here:  This portion of the energy evaluation functions
     is currently unimplemented.  */
  /* double **override; */

  /* Discard any information after a comment marker */
  i = 0;
  while (line[i] != '#' && line[i] != '\0') { i++; }
  if ( line[i] == '#' ) { line[i] = '\0'; }

  /* Search for the end of config file line, return when found */
  if (strncmp("END", line, strlen("END")) == 0) {
    fprintf (stderr, "Found the end of the file, returning\n");
    if ( term_index == 0 ) { return 0; }
    else { return 1; }
  }

  /* Make certain we haven't read too many terms for our defined
     maximum force field.  This needs to be done dynamically,
     eventually.  */

  if ( term_index >= MAX_TERMS ) {
    error_exit("Too many terms found in force field configuration "
        "file, make a simpler force field, or consider "
        "developing a dynamic force field array");
  }

  if (strncmp("global", line, strlen("global")) == 0) {
    printf ("Configuring global parameters from file.\n");

    /* Do whatever global definitions are needed here.  Note that
     this variable is _truly_ global.  This is the _only_ function
     that should create this variable.  Note that _all_ other
     access must be done with the global pointer to this
     variable.  Finally, in early development, there is no use for
     this variable, so it is not implemented.  It is meant to hold
     information such as the simulation space, mean fields on the
     system (if they're not implemented within their own energy
     function), and other similar parameters */

  } else if(strncmp("bond_gen_dreiding2", line,
          strlen("bond_gen_dreiding2")) == 0) {

    char *full_command;

    full_command = get_full_command(cfg_file, line);

    /* Gather relevant options */
    init_flags = INIT_NRG_FUNC | get_init_flags(full_command);

    /* And initialize the function with the init flags we found */
    funclist[term_index] = bond_gen_dreiding2;
    funclist[term_index](molecule_list, init_flags, NULL, NULL);
    term_index++;

    free(full_command);

  } else if(strncmp("bend_gen_dreiding2", line,
          strlen("bend_gen_dreiding2")) == 0) {

    char *full_command;

    full_command = get_full_command(cfg_file, line);

    /* Gather relevant options */
    init_flags = INIT_NRG_FUNC | get_init_flags(full_command);

    /* And initialize the function with the init flags we found */
    funclist[term_index] = bend_gen_dreiding2;
    funclist[term_index](molecule_list, init_flags, NULL, NULL);
    term_index++;

    free(full_command);

  } else if(strncmp("torsion_bouldergroup1", line,
          strlen("torsion_bouldergroup1")) == 0) {

    char *full_command;

    full_command = get_full_command(cfg_file, line);

    /* Gather relevant options */
    init_flags = INIT_NRG_FUNC | get_init_flags(full_command);

    /* And initialize the function with the init flags we found */
    funclist[term_index] = torsion_bouldergroup1;
    funclist[term_index](molecule_list, init_flags, NULL, NULL);
    term_index++;

    free(full_command);

  } else if(strncmp("inv_gen_dreiding2", line,
          strlen("inv_gen_dreiding2")) == 0) {

    char *full_command;

    full_command = get_full_command(cfg_file, line);

    /* Gather relevant options */
    init_flags = INIT_NRG_FUNC | get_init_flags(full_command);

    /* And initialize the function with the init flags we found */
    funclist[term_index] = inv_gen_dreiding2;
    funclist[term_index](molecule_list, init_flags, NULL, NULL);
    term_index++;

    free(full_command);

  } else if(strncmp("vdw_bouldergroup1", line,
          strlen("vdw_bouldergroup1")) == 0) {

    char *full_command;

    full_command = get_full_command(cfg_file, line);
```

```
    /* Gather relevant options */
    init_flags = INIT_NRG_FUNC | get_init_flags(full_command);

    /* And initialize the function with the init flags we found */
    funclist[term_index] = vdw_bouldergroup1;
    funclist[term_index](molecule_list, init_flags, NULL, NULL);
    term_index++;

    free(full_command);

  } else if(strncmp("coul_rawsum", line,
            strlen("coul_rawsum")) == 0) {

    char *full_command;

    full_command = get_full_command(cfg_file, line);

    /* Gather relevant options */
    init_flags = INIT_NRG_FUNC | get_init_flags(full_command);

    /* And initialize the function with the init flags we found */
    funclist[term_index] = coul_rawsum;
    funclist[term_index](molecule_list, init_flags, NULL, NULL);
    term_index++;

    free(full_command);

  }

  /* Finally, when we're done, read the next line */
  line_p = fgets(line, MAX_FLINE, cfg_file);

  }

  /* Before returning, make certain to re-initialize all of the energy
   functions in the funclist[] array with the global_sim_parms */
  funclist[term_index] = NULL;

  return yes;

}

/* The following function extracts the full command name from an
   already open initialization file.  Warning!  This function returns
   newly allocated space, which must be freed later.  */
static char* get_full_command(FILE *infile, char *initial_line) {

  char *full_command;
  boolean need_more_lines = 1;
  int i;

  full_command = malloc(sizeof(char) * strlen(initial_line) + 1);

  strcpy(full_command, initial_line);

  for(i = 0; i < strlen(full_command); i++ ) {
  if (full_command[i] == ';') { need_more_lines = no; break; }
  }

  /* Remove any newlines from the existing full_command.  Note
     that octal 015 and 012 are the most common newlines used by
     various OS's. */
  for(i = 0; i < strlen(full_command); i++) {
  if (full_command[i] == 015 || full_command[i] == 012) {
    full_command[i] = ' ';
  }
  }

  if (need_more_lines) {
  /* Read characters until the semicolon is encountered. */
  /* Note that on some systems, EOF == -1.  In these cases, */
  /* it is likely that a char cannot represent that value.  */
  /* As a result, we should always use int for reading characters */
  /* from files */
  int c;

  while( (c = getc(infile)) != ';' && c != EOF ) {
    if( c == 015 || c == 012 ) { c = ' '; }
    full_command = realloc(full_command, sizeof(char) +
              strlen(full_command) + 2);
    if(!full_command) {
    fprintf(stderr, "Unable to expand the size of "
        "full_command in initialization of .ff_form "
        "file, this is a critical error, exiting\n");
    }

    full_command[strlen(full_command)] = c;
    full_command[strlen(full_command) + 1] = '\0';
  }

  if (c == EOF) {
    fprintf(stderr, "Reached end of file while trying to read "
        "options.  This is a critical file error, "
        "exiting ...\n");
    exit(0);
  }
  }
  return full_command;
}

/* The following function returns the init flags from the provided
   command.  If one is using it to initialize, it should be bitwise
   or'ed with INIT_NRG_FUNC.  It's one big messy parsing function. */
static long int get_init_flags(char *command) {

  int i = 0, paren_depth = 0;
  long int return_value = 0;
  char token[MAX_STR];
  boolean in_first_term = no;

  /* Fast forward to opening parenthesis */
  while(command[i] != '\0' && command[i] != '(') { i++; }

  in_first_term = yes;

  /* Process rest of command */
  while (command[i] != '\0' ) {
  char c = command[i];
  int j = 0;

  /* Begin with handling parentheses */
  if(c == '(') { paren_depth++; i++; continue; }
  if(c == ')') {
    if(in_first_term) { in_first_term = no; }
    paren_depth--;
    i++; continue;
  }

  /* Discard whitespace characters */
  while (c != '\0' && ( c == ' ' || c == '\t' || c == ',' ) ) {
    i++;
    c = command[i];
  }
  if(c == '\0') { break; }

  /* Get the next token */
  while (c != '\0' && c != ' ' && c != '\t' && c != ',' && c != '('
      && c != ')') {
    token[j] = c;
    j++;
    i++;
    c = command[i];
  }
  token[j] = '\0';

  /* This is where we handle the tokens.  Each token must have a
     case for handling explicitly.  This means that token names must
     be unique (across functions), though this should not be an
     issue since tokens are #define'd in the header file.  */
  if(in_first_term) {
    if ( !strcmp(token, "OMIT12") ) {
    return_value = return_value | OMIT12;
    } else if ( !strcmp(token, "OMIT13") ) {
    return_value = return_value | OMIT13;
    } else if ( !strcmp(token, "OMIT14") ) {
    return_value = return_value | OMIT14;
    } else if ( !strcmp(token, "UPDATE_FORCES") ) {
    return_value = return_value | UPDATE_FORCES;
    }
  }
  }

  if(paren_depth != 0) {
  fprintf(stderr, "Command \"%s\" has unbalanced parentheses, "
      "cannot process.  Please repair your input and try "
      "again ... exiting\n", command);
  exit(0);
  }

  return return_value;
}

static void initialize_forces_and_velocities(atom *this_molecule) {

  printf("This is a stub function.  When needed, it should initialize other[1]
for force vectors, and other[3] for velocity vectors for each atom in the
supplied molecule\n");

  return;
}

/* The following function is the first of the implementations of this
   library, and evaluates the energy of the system, and assumes the
   library has been properly initialized already */

long double get_system_energy( atom **atom_list, long flags) {

  long double total_energy = 0;
  int term_index = 0;

  printf("\nIn get_system_energy(), about to loop through functions in
funclist[]\n");

  while ( funclist[term_index] ) {
  total_energy += funclist[term_index](atom_list, flags, NULL, parms);
  term_index++;
  }

  return total_energy;

}

/* The following function implements generic dreiding2 bond stretching.
   See options for it in nrgfunc.h */
long double bond_gen_dreiding2(atom **atom_list, long flags,
                double* override, global_sim_parms* parms) {

  long double this_energy = 0.0;
  atom *this_molecule;
  int member_index = 0;

  /* If we're initializing, _only_ do initialization things, then
    simply return.  In this case, we need to initialize the bond list
    in all of the provided molecules.  For this task, we are doing
    essentially the same thing that print_molecule_connectivity() is
    doing, though the function actually outputs to a file stream.
    Instead of creating a temporary file, and using it, I'll re-write
    the function in atom_handling.c to use a new function to generate
    the list, then simply output that information. */

  if ( INIT_NRG_FUNC & flags ) {

  while (atom_list[member_index]) {

    int *connectivity, conn_length, i;
    double *special_index;
    atom *this_atom;
```

```
    /* Get the base of the molecule (we reuse it often), and get a
     connectivity list as well */
    this_molecule = molecule_return_base(atom_list[member_index]);
    connectivity = generate_connectivity_list(this_molecule);

    /* Now, for each atom in the list passed to us, we need to
     assign the proper connectivity.  This way, connectivity will
     be acessible from any particular atom in a given molecule */

    for ( this_atom = this_molecule; this_atom != NULL;
      this_atom = this_atom->next ) {
    this_atom->CONN = connectivity;
    }

    /* The following demonstrates how to loop through all of the
     connectivity on some atom */
#if 0
    {
    for( i = 0; CONNV(this_molecule, i) != 0 ||
         CONNV( this_molecule, i + 1 )  != 0; i += 2 ) {
      printf("Bond: %d %d\n", CONNV(this_molecule, i) + 1,
         CONNV(this_molecule, i + 1) + 1 );
    }
    }
#endif

    /* Now, we need to build the 'special index' for lookups when we
     evaluate the energy. The special index will be parallel to
     the connectivity list.  It will be a 1-D array that contains
     the equilibrium bond length (in Angstroms), and the energy
     cost costant for that bond (completely paralell to the
     connectivity array).  Note that the equilibrium bond length
     is taken from the original structure.  The original structure
     must be pre-optimized for this section to work.  */

    conn_length = 0;
    while( connectivity[conn_length] != 0 ||
       connectivity[conn_length + 1] != 0 ) { conn_length += 2; }

    special_index = malloc( (conn_length + 2 ) * sizeof(double) );
    if (special_index == NULL) {
    error_exit("Unable to allocate memory for special index in "
          "bond_gen_drieding2");
    }

    /* Now, populate the array */
    for ( i = 0; i < conn_length; i += 2 ) {

    float this_bond_order;

    special_index[i] =
      get_interatomic_distance( this_molecule + connectivity[i],
                this_molecule + connectivity[i + 1]);

    /* And for the dreiding 2 force field, the bond energy is one of
     several values, based on the type of bond.  The units on all
     values are in kcal/(molecule angstrom^2)  */
    this_bond_order =
      get_stored_bond_order( this_molecule + connectivity[i],
               this_molecule + connectivity[i + 1]);
    if ( this_bond_order == 1 ) {
      special_index[i + 1] = 700 / AVAG_NUMBER;
    } else if ( this_bond_order == 1.5 ) {
      special_index[i + 1] = 1050 / AVAG_NUMBER;
    } else if ( this_bond_order == 2 ) {
      special_index[i + 1] = 1400 / AVAG_NUMBER;
    } else if ( this_bond_order == 3 ) {
      special_index[i + 1] = 2100 / AVAG_NUMBER;
    }
    }
    /* Set the final two members of special index to 0, as flag
     values */
    special_index[conn_length] = special_index[conn_length + 1] = 0;

    /* And make the whole molecule point to this new special index */
    for ( this_atom = this_molecule; this_atom != NULL;
      this_atom = this_atom->next ) {
    this_atom->BOS1I = special_index;
    }

    /* Finally, if we were also asked to UPDATE_FORCES, then we
     should make that call here as well */
    if ( UPDATE_FORCES & flags ) {
    initialize_forces_and_velocities(this_molecule);
    }

    member_index++;
  }

  /* The connectivity (bond list) and energy lookup list for this
     atom is finished.  */

  return 0.0;
  }


  /* For each molecule in the atom list, evaluate the energy */
  while ( atom_list[member_index] ) {

  if ( flags & RETURN_ENERGY ) {

    int i;
    double bdev;

    this_molecule = atom_list[member_index];

    for( i = 0; CONNV(this_molecule, i) != 0 ||
       CONNV( this_molecule, i + 1 )  != 0; i += 2 ) {

    /* The following number may be positive or negative, but since
       we're taking the square of it, it doesn't matter */
    bdev = get_interatomic_distance(
           &this_molecule[CONNV(this_molecule, i)],
         &this_molecule[CONNV(this_molecule, i + 1)] ) -
```

```
       BOS1IV(this_molecule, i);

       this_energy += bdev * bdev * BOS1IV(this_molecule, i + 1);
     }

     /* Evaluate and update forces if requested */
     if ( flags & UPDATE_FORCES ) {
     error_exit("Updating forces in bond_gen_dreiding2 has not been "
            "implemented yet");
     }

   }
  member_index++;
  }
  printf("In bond_gen_dreiding2(), returning %Lg\n", this_energy);

  return this_energy;
}

/* The following function implements generic dreiding2 bond bending.
   See options for it in nrgfunc.h */
long double bend_gen_dreiding2(atom **atom_list, long flags,
                double* override, global_sim_parms* parms) {

  long double this_energy = 0.0;
  atom *this_molecule;
  int member_index = 0;

  /* If we're initializing, _only_ do initialization things, then
   simply return.  In this case, we need to initialize the angle list
   in all of the provided molecules.  This is another capability
   that belongs in atom_handling.c.  */

  if ( INIT_NRG_FUNC & flags ) {

  while (atom_list[member_index]) {

    int *angles /*, list_length, i */;
    /*double *special_index; */
    atom *this_atom;

    /* Get the base of the molecule (we reuse it often), and get a
     connectivity list as well */
    this_molecule = molecule_return_base(atom_list[member_index]);

    /* The following function should refer to something like
     generate_angle_list.  */
    angles = generate_angle_list(this_molecule);

    /* Now, for each atom in the list passed to us, we need to
     assign the proper angles.  This way, the list of angles will
     be acessible from any particular atom in a given molecule */

    for ( this_atom = this_molecule; this_atom != NULL;
        this_atom = this_atom->next ) {
        this_atom->ANGLE_LIST = angles;
    }

    /* The following demonstrates how to loop through all of the
     connectivity on some atom */
#if 0
    {
    int i;
    for( i = 0; ANGLEV(this_molecule, i) != 0 ||
         ANGLEV( this_molecule, i + 1 )  != 0 ||
         ANGLEV( this_molecule, i + 2 )  != 0; i += 3 ) {
      printf("Angle: %d %d %d\n", ANGLEV(this_molecule, i) + 1,
         ANGLEV(this_molecule, i + 1) + 1, ANGLEV(this_molecule, i + 2) + 1 );
    }
    }
#endif

    printf("The next step in development is to generate the special index to
find all of the equilibrium bond angles.  This will require developing a _fast_
way of evaluating the angles (before we even build the list).  At this point,
the actual work of this function is nearly finished\n");

    /* Now, we need to build the 'special index' to look up the
     original equilibrium bond angles.  Note that in the dreiding
     2 force field, absolutely every angle has the same force
     constant */

    /*    conn_length = 0;
        while( connectivity[conn_length] != 0 ||
        connectivity[conn_length + 1] != 0 ) { conn_length += 2; }

        special_index = malloc( (conn_length + 2 ) * sizeof(double) );
        if (special_index == NULL) {
        error_exit("Unable to allocate memory for special index in "
        "bond_gen_drieding2");
        } */

    /* Now, populate the array */
    /*    for ( i = 0; i < conn_length; i += 2 ) {

    float this_bond_order;

    special_index[i] =
    get_interatomic_distance( this_molecule + connectivity[i],
    this_molecule + connectivity[i +
    1]);
    } */

    /* Set the final two members of special index to 0, as flag
     values */
    /*    special_index[conn_length] = special_index[conn_length +
        1] = 0; */

    /* And make the whole molecule point to this new special index */
    /*    for ( this_atom = this_molecule; this_atom != NULL;
        this_atom = this_atom->next ) {
        this_atom->BOS1I = special_index;
        } */
```

```
    /* Finally, if we were also asked to UPDATE_FORCES, then we
     should make that call here as well */
    if ( UPDATE_FORCES & flags ) {
    initialize_forces_and_velocities(this_molecule);
    }

    member_index++;
  }


  /* The last task is to make the atoms atom_list point to the
     _base_ of each molecule.  This will speed up the execution of
     the energy evaluation. */
  member_index = 0;
  while ( atom_list[member_index] ) {
    while( atom_list[member_index]->previous != NULL ) {
    atom_list[member_index] = atom_list[member_index]->previous;
    }
    member_index++;
  }

  return 0.0;
  }


  /* For each molecule in the atom list, evaluate the energy */
  while ( atom_list[member_index] ) {

  if ( flags & RETURN_ENERGY ) {

    /*    int i;
        double bdev; */

    this_molecule = atom_list[member_index];

    /*    for( i = 0; CONNV(this_molecule, i) != 0 ||
        CONNV( this_molecule, i + 1 )  != 0; i += 2 ) { */

    /* The following number may be positive or negative, but since
      we're taking the square of it, it doesn't matter */
    /*    bdev = get_interatomic_distance(
                  &this_molecule[CONNV(this_molecule, i)],
                  &this_molecule[CONNV(this_molecule, i + 1)] ) -
      BOS1IV(this_molecule, i);

      this_energy += bdev * bdev * BOS1IV(this_molecule, i + 1);
      } */

    /* Evaluate and update forces if requested */
    if ( flags & UPDATE_FORCES ) {
    error_exit("Updating forces in bond_gen_dreiding2 has not been "
           "implemented yet");
    }

    member_index++;
  }
  }


  printf("In bend_gen_dreiding2(), returning %Lg\n", this_energy);


  return this_energy;
}

/* The following function implements the boulder groups handling of
   torsions.  See options for it in nrgfunc.h */
long double torsion_bouldergroup1(atom **atom_list, long flags,
             double* override, global_sim_parms* parms) {

  long double this_energy = 0.0;

  printf("In torsion_bouldergroup1(), returning %Lg\n", this_energy);


  return this_energy;
}

/* The following function implements generic dreiding2 inversion
   (umbrella stretching).  See options for it in nrgfunc.h */
long double inv_gen_dreiding2(atom **atom_list, long flags,
             double* override, global_sim_parms* parms) {

  long double this_energy = 0.0;

  printf("In inv_gen_dreiding2(), returning %Lg\n", this_energy);


  return this_energy;
}

/* The following function implements the boulder group's handling of
   van der Waals parameters.  See options for it in nrgfunc.h */
long double vdw_bouldergroup1(atom **atom_list, long flags,
             double* override, global_sim_parms* parms) {

  static long int sim_flags = 0;
  long double this_energy = 0.0;

  /* If we're in an initialization pass, set our variables */
  if ( INIT_NRG_FUNC & flags ) {
  sim_flags = flags;
  }

  printf("In vdw_bouldergroup1(), returning %Lg\n", this_energy);
  printf("Flags is: %ld\n", sim_flags);


  return this_energy;
}

/* The following function implements a generic coulumbic energy.  See
   options for it in nrgfunc.h */
long double coul_rawsum(atom **atom_list, long flags,
             double* override, global_sim_parms* parms) {

  static long int sim_flags = 0;
  long double this_energy = 0.0;

  /* If we're in an initialization pass, set our variables */
  if ( INIT_NRG_FUNC & flags ) {
  sim_flags = flags;
  }

  printf("In coul_rawsum(), returning %Lg\n", this_energy);
  printf("Flags is: %ld\n", sim_flags);

  return this_energy;
}
```